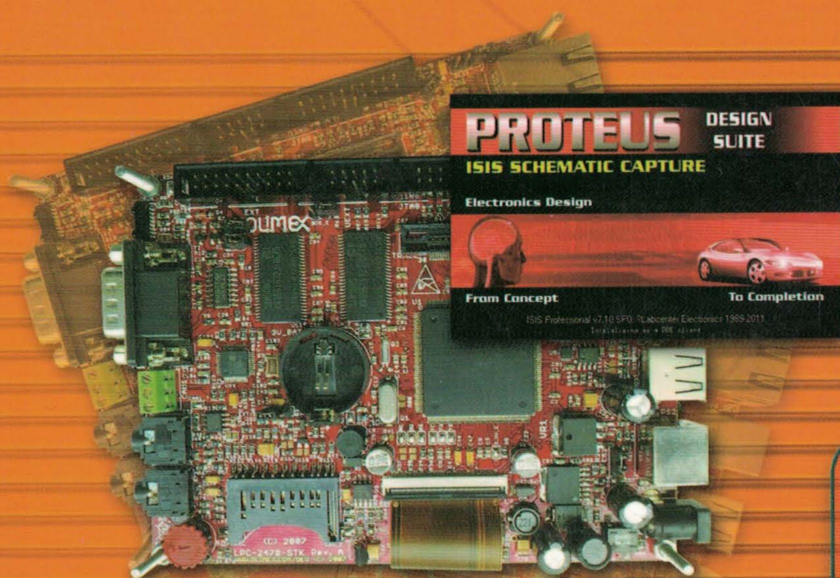
ARM嵌入式系统基础与应用

—基于Proteus和IAR EWARM

徐爱钧　编著





随书配套光盘包含大量实例和代码

- Proteus for ARM 虚拟仿真平台
- IAR EWARM 开发软件
- NXP ARM 处理器原理
- LPC2100 系列内部资源与外部扩展应用
- μC/OS Ⅱ 嵌入式操作系统应用





机械工业出版社
CHINA MACHINE PRESS

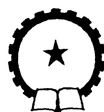
VISIT…

LANZAROTE Caliente.com

ARM 嵌入式系统基础与应用

——基于 Proteus 和 IAR EWARM

徐爱钧　编著



机 械 工 业 出 版 社

本书以 NXP 公司的 LPC2100 系列 ARM 处理器为原型、IAR EWARM 为软件平台，阐述了基于 Proteus 虚拟仿真技术的 ARM 嵌入式应用开发技术。分析了与 ARM 处理器架构相关的技术要点，详细介绍了 LPC2138 ARM 处理器片内功能资源和外部扩展应用技术，给出了大量 Proteus 虚拟仿真实例，并配有包含 Proteus DEMO 版软件包和书中全部实例的光盘。

本书适合于从事 ARM 嵌入式系统开发设计的工程技术人员阅读，也可作为大专院校相关专业嵌入式系统课程的教学用书。

图书在版编目（CIP）数据

ARM 嵌入式系统基础与应用：基于 Proteus 和 IAREWARM / 徐爱钧编著.
—北京：机械工业出版社，2013.10
ISBN 978-7-111-44508-1

Ⅰ．A…　Ⅱ．①徐…　Ⅲ．①微处理器－系统设计　Ⅳ．①TP332

中国版本图书馆 CIP 数据核字（2013）第 251192 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：时　静
责任编辑：时　静
责任印制：李　洋

北京瑞德印刷有限公司印刷（三河市胜利装订厂装订）

2013 年 11 月第 1 版 • 第 1 次印刷
184mm×260mm • 22.75 印张 • 560 千字
0001－3500册
标准书号：ISBN 978-7-111-44508-1
ISBN 978-7-89405-106-6（光盘）
定价：56.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务　　网络服务
社服务中心：（010）88361066　　教材网：http://www.cmpedu.com
销售一部：（010）68326294　　机工官网：http://www.cmpbook.com
销售二部：（010）88379649　　机工官博：http://weibo.com/cmp1952
读者购书热线：（010）88379203　　封面无防伪标均为盗版

前　　言

随着 ARM 处理器在嵌入式系统中的应用日益广泛，各种 ARM 开发技术层出不穷。英国 Labcenter 公司推出的 Proteus 软件是一款极好的 ARM 处理器开发平台，它以其特有的虚拟仿真技术很好地解决了 ARM 处理器及其外围电路的设计和协同仿真问题，可以在没有 ARM 处理器实际硬件的条件下，利用 PC 进行虚拟仿真，实现 ARM 应用系统的软、硬件协同设计。采用 Proteus 虚拟仿真技术，可以在原理图设计阶段，对系统性能进行评估，验证所设计的电路是否达到技术指标要求，使设计过程变得简单容易。

Proteus 软件已有 20 多年的历史，涵盖了 PIC、AVR、MCS8051、68HC11 和 ARM 等微处理器模型，以及多种常用电子元器件，包括 74 系列、CMOS4000 系列集成电路、A-D 和 D-A 转换器、键盘、LCD 显示器和 LED 显示器，还提供示波器、逻辑分析仪、通信终端、电压/电流表、I^2C/SPI 终端等各种虚拟仪表，这些都可以直接用于虚拟仿真，极大地提高了应用系统的设计效率。

利用 Proteus 软件平台来学习 ARM 嵌入式系统应用开发技术，可以使学习过程变得直观形象。基于原理图的虚拟模型仿真，可实现源代码的程序调试，还能看到程序运行后的输入输出效果。在 PC 上修改原理电路图要比修改实际硬件电路容易得多，成功进行虚拟仿真并获得期望结果之后，再制作硬件电路板，进行在线调试，可以获得事半功倍的效果。

本书在构思及选材上，以易学易用为原则，详细介绍了 NXP 公司的 LPC2138ARM 处理器片内功能资源和外部扩展应用技术，结合 IAR EWARM 开发软件，给出了大量 Proteus 虚拟仿真实例。全书共分 6 章，各章主要内容如下：

第 1 章　ARM 体系结构基础。介绍 ARM 支持的数据类型和存储器结构、ARM 处理器的工作状态和运行模式、寻址方式、指令集、ARM 汇编语言规范，给出了用汇编语言编写的启动程序和其他范例。

第 2 章　Proteus for ARM7 虚拟仿真。介绍了 Proteus 支持的 ARM 模型、ELF/SWARF 装载器、在 ISIS 集成环境中绘制原理电路图及源代码仿真调试，以及采用 IAR EWARM 编译环境生成带有仿真调试信息可执行文件的方法。

第 3 章　LPC213x ARM 处理器。介绍了 ARM 处理器特性与存储器结构、系统控制模块、向量中断控制器 VIC、引脚功能配置等。

第 4 章　LPC2138 片内集成功能应用技术。介绍了通用输入输出端口 GPIO、通用异步接收发送器 UART、I^2C 接口、SPI 接口、SSP 接口、定时器/计数器、脉宽调制器 PWM、A-D 和 D-A 转换器、实时时钟和看门狗定时器，并给出了具体应用实例。

第 5 章　LPC2138 片外扩展功能应用技术。介绍了点阵字符型和点阵图型液晶显示模块接口技术、用 DS18B20 与 LPC2138 实现的数字温度计、用 DS1302 与 LPC2138 实现的万年历、SD 卡与 LPC2138 的接口及应用、网络芯片 ENC28J60 与 LPC2138 的接口及应用。

第 6 章　μC/OS-II 在 LPC2138 上的移植与应用。介绍了 μC/OS-II 在 LPC2138 上的移植、

μC/OS-II 的任务管理、任务的同步与通信、信号量、消息邮箱，以及编写 μC/OS-II 应用程序的方法。

附录 A 介绍了广州风标电子技术有限公司生产的 ARM 嵌入式硬件开发平台，给出了主要原理电路以及具体应用方法。

附录 B 介绍了 Proteus 中的常用元器件。

本书每章均给出了大量应用实例，所有实例都在 Proteus 平台上仿真通过，并随本书的配套光盘提供给读者，对加深理解 LPC2138ARM 处理器基本原理，提高应用设计能力具有极大帮助。

本书在写作过程中得到广州风标电子技术有限公司（http://www.windway.cn）匡载华总经理的大力支持和热情帮助，还得到彭秀华、徐阳、范林、肖恩凯、郑玉章等的协助，在此一并表示感谢。

由于作者水平有限，书中难免会有错误和不妥之处，恳请广大读者批评指正，读者可通过电子邮件：ajxu@tom.com、ajxu41@sohu.com 直接与作者联系。

Proteus 的 DEMO 软件可到官方网站 http://www.labcenter.co.uk 下载，或者与它的国内代理商广州风标电子技术有限公司联系购买正版软件。

徐爱钧　于长江大学

2013 年 8 月

目　　录

前言
第 1 章　ARM 体系结构基础 …… 1
1.1　ARM 的数据类型和存储器格式 …… 1
1.2　ARM 处理器的工作状态和运行模式 …… 2
1.3　寄存器组织 …… 3
1.3.1　寄存器分类 …… 3
1.3.2　通用寄存器 …… 5
1.3.3　程序状态寄存器 …… 6
1.4　异常 …… 8
1.4.1　ARM 体系结构所支持的异常类型 …… 8
1.4.2　各类异常的具体描述 …… 9
1.4.3　对异常的响应和返回 …… 11
1.5　ARM 指令集 …… 12
1.5.1　ARM 指令的功能与格式 …… 12
1.5.2　指令的条件码 …… 13
1.6　ARM 指令的寻址方式 …… 14
1.6.1　寄存器寻址 …… 14
1.6.2　立即寻址 …… 14
1.6.3　寄存器移位寻址 …… 15
1.6.4　寄存器间接寻址 …… 15
1.6.5　基址寻址 …… 16
1.6.6　相对寻址 …… 16
1.6.7　多寄存器寻址 …… 16
1.6.8　堆栈寻址 …… 17
1.6.9　块拷贝寻址 …… 17
1.6.10　ARM 伪指令 …… 18
1.7　Thumb 指令集 …… 18
1.8　ARM 汇编语言编程 …… 19
1.8.1　ARM 汇编语言规范 …… 19
1.8.2　汇编伪指令 …… 20
1.8.3　程序设计举例 …… 20
1.8.4　汇编语言与 C/C++混合编程 …… 22
1.9　启动代码 …… 22

1.9.1 Startup.s 文件 …… 23
1.9.2 Target.c 文件 …… 25
1.9.3 Target.h 文件 …… 26
1.9.4 Config.h 文件 …… 27
复习思考题 …… 27
第 2 章 Proteus for ARM7 虚拟仿真 …… 29
2.1 Proteus for ARM 简介 …… 29
2.1.1 Proteus 支持的 ARM 模型 …… 29
2.1.2 Proteus ELF/SWARF 装载器 …… 31
2.1.3 Proteus LPC2000 的调试窗口 …… 31
2.2 集成环境 ISIS …… 35
2.3 原理图绘制与源代码仿真调试 …… 38
2.4 用 IAR EWARM 生成源代码仿真调试程序 …… 42
2.4.1 创建项目 …… 42
2.4.2 项目选项配置与编译链接 …… 45
复习思考题 …… 49
第 3 章 LPC213x ARM 处理器 …… 50
3.1 LPC213x 处理器特性与存储器结构 …… 50
3.1.1 主要特性 …… 50
3.1.2 存储器结构 …… 51
3.1.3 存储器重新映射和 Boot Block …… 53
3.1.4 存储器加速模块 …… 55
3.2 系统控制模块 …… 57
3.2.1 时钟频率控制 …… 57
3.2.2 VPB 分频器 …… 61
3.2.3 功率控制 …… 62
3.2.4 复位 …… 63
3.2.5 唤醒定时器 …… 64
3.3 外部中断输入 …… 65
3.4 向量中断控制器 VIC …… 68
3.4.1 VIC 寄存器 …… 68
3.4.2 中断源 …… 71
3.4.3 VIC 使用注意事项 …… 73
3.5 引脚功能配置 …… 74
3.5.1 引脚选择寄存器 …… 74
3.5.2 引脚配置示例 …… 76
3.6 系统控制应用举例 …… 76
3.6.1 存储器映射 …… 76
3.6.2 锁相环 …… 78

3.6.3 存储器加速模块 83
3.7 VIC 中断应用举例 84
3.7.1 外部中断 84
3.7.2 向量中断 86
3.7.3 嵌套中断 88
3.7.4 快速中断 90
3.7.5 软件中断 91
复习思考题 93
第 4 章 LPC2138 片内集成功能应用技术 94
4.1 通用输入输出端口 GPIO 94
4.1.1 主要特性 94
4.1.2 寄存器描述 94
4.1.3 应用举例 95
4.2 通用异步接收发送器 UART 100
4.2.1 主要特性 100
4.2.2 寄存器描述 101
4.2.3 应用举例 107
4.3 I^2C 接口 114
4.3.1 主要特性 114
4.3.2 寄存器描述 117
4.3.3 应用举例 120
4.4 SPI 接口 124
4.4.1 主要特性 124
4.4.2 SPI 数据传输 125
4.4.3 寄存器描述 126
4.4.4 应用举例 128
4.5 SSP 接口 133
4.5.1 主要特性 133
4.5.2 寄存器描述 133
4.5.3 应用举例 136
4.6 定时器/计数器 139
4.6.1 主要特性 139
4.6.2 寄存器描述 139
4.6.3 应用举例 144
4.7 脉宽调制器 PWM 152
4.7.1 主要特性 152
4.7.2 寄存器描述 154
4.7.3 应用举例 158
4.8 A-D 转换器 167

4.8.1 主要特性 167
4.8.2 寄存器描述 168
4.8.3 应用举例 170
4.9 D-A 转换器 177
4.9.1 主要特性 177
4.9.2 寄存器描述 177
4.9.3 应用举例 178
4.10 实时时钟 183
4.10.1 主要特性 183
4.10.2 寄存器描述 183
4.10.3 应用举例 187
4.11 看门狗定时器 192
4.11.1 主要特性 192
4.11.2 寄存器描述 193
4.11.3 应用举例 194
复习思考题 199
第 5 章 LPC2138 片外扩展功能应用技术 201
5.1 液晶显示器 LCD 接口技术 201
5.1.1 点阵字符型液晶显示模块接口技术 201
5.1.2 12864 点阵图型液晶显示模块接口技术 210
5.1.3 T6963C 点阵图型液晶显示模块接口技术 219
5.2 用 DS18B20 与 LPC2138 实现的数字温度计 228
5.3 用 DS1302 与 LPC2138 实现的万年历 236
5.4 SD 卡与 LPC2138 的接口及应用 249
5.4.1 SD 卡简介 249
5.4.2 FAT16 文件系统 251
5.4.3 SD 卡接口应用举例 253
5.5 网络芯片 ENC28J60 与 LPC2138 的接口及应用 269
5.5.1 ENC28J60 的功能模块与引脚分布 269
5.5.2 ENC28J60 主要特性 271
5.5.3 ENC28J60 应用举例 272
复习思考题 310
第 6 章 μC/OS-II 在 LPC2138 上的移植与应用 311
6.1 μC/OS-II 简介 311
6.2 μC/OS-II 在 LPC2138 上的移植 312
6.2.1 编写 OS_CPU.H 文件 313
6.2.2 编写 OS_cpu_c.c 文件 314
6.2.3 编写 Os_cpu_asm 文件 317
6.3 编写μC/OS-II 应用程序 319

6.3.1 任务管理 …… 319
6.3.2 任务管理应用编程举例 …… 322
6.4 任务的同步与通信 …… 325
6.4.1 信号量 …… 325
6.4.2 信号量应用编程举例 …… 328
6.4.3 消息邮箱 …… 330
6.4.4 消息邮箱应用编程举例 …… 333
复习思考题 …… 335
附录 …… 337
附录 A ARM 硬件开发平台 …… 337
A.1 电源电路 …… 337
A.2 复位电路 …… 338
A.3 时钟电路 …… 338
A.4 RS232 串口和 MODEM 电路 …… 339
A.5 键盘电路 …… 340
A.6 LED 显示电路 …… 340
A.7 LCD 显示电路 …… 342
A.8 I^2C 接口电路 …… 342
A.9 音频接口电路 …… 343
A.10 电动机驱动电路 …… 343
A.11 ADC 采样电路 …… 345
A.12 网络接口电路 …… 345
A.13 SD 卡接口电路 …… 346
A.14 USB 接口电路 …… 346
A.15 总线接口电路 …… 347
A.16 PS/2 接口电路 …… 347
A.17 扩展模块接口电路 …… 348
A.18 JTAG 接口电路 …… 348
附录 B Proteus 中的常用元器件 …… 350
参考文献 …… 352

第 1 章　ARM 体系结构基础

ARM 是 Advanced RISC Machines 的缩写，ARM 公司专注于设计，本身不生产芯片，靠转让设计许可由合作伙伴公司来生产基于 ARM 核的处理器芯片。目前，ARM 公司在世界范围内的合作伙伴已超过 100 个，它们生产的 ARM 核芯片各具特色，在移动通信、手持计算、多媒体应用以及工业测量控制等领域得到了广泛应用。ARM 的 32 位 RISC 体系结构目前被公认为业界领先，所有 ARM 核处理器共享这一体系结构，从而确保应用人员在软件开发上可以获得最大的回报。ARM 体系结构采用指令流水线，分别为取指、译码和执行，而 ARM9 体系结构则采用 5 级指令流水线，分别为取指、译码、执行、缓存和写回。利用流水线重叠技术使系统性能大为提高。

1.1　ARM 的数据类型和存储器格式

ARM 体系结构支持三种数据类型：字（Word），长度为 32 位；半字（Halfword），长度为 16 位；字节（Byte），长度为 8 位。

在 ARM 存储器组织中，半字必须与 2 字节边界对齐，字必须与 4 字节边界对齐，即半字必须开始于偶数地址，字必须开始于 4 的倍数的字节地址，如图 1-1 所示。

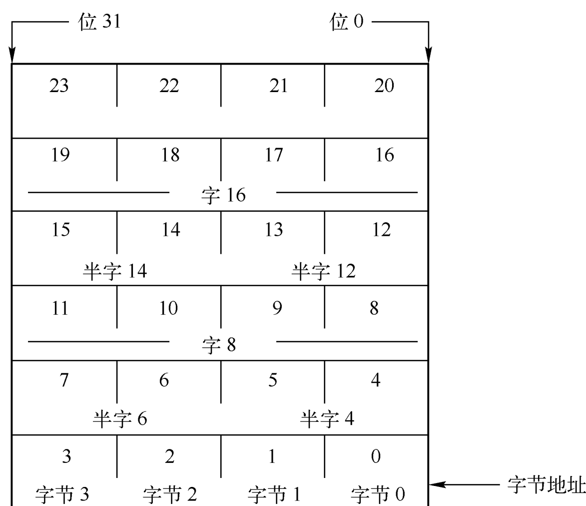


图 1-1　ARM 数据类型存储图

ARM 处理器采用冯·诺依曼（Von Neumann）形式的存储器结构，使用 2^{32} 个字节的单一、线性地址空间。从 0 字节开始到第 3 字节存放第一个字数据，从第 4 字节到第 7 字节存放第二个字数据，依次类推。指令和数据共用一条 32 位数据总线，只有加载、保存和交换

指令可以访问存储器中的数据。作为 32 位的微处理器，ARM 体系结构所支持的最大地址空间为 4GB（2^{32}B）。

ARM7 处理器采用 3 级指令流水线，在执行当前指令的同时对下一条指令进行译码，并对再下一条指令进行指令预取，如果预取操作溢出了地址空间顶端，则不会产生执行动作，并导致不可预知的结果。

ARM 体系结构可以用两种方法存储字数据，分别称为大端格式（Big Endian）和小端格式（Little Endian）。在大端格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中，如图 1-2 所示。

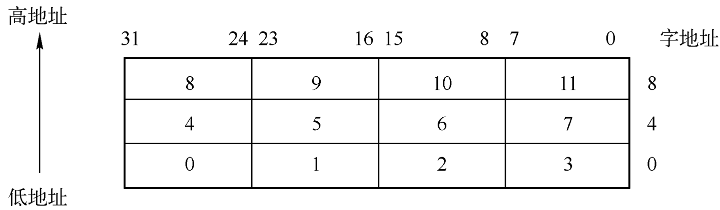


图 1-2　大端存储格式

与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节，如图 1-3 所示。

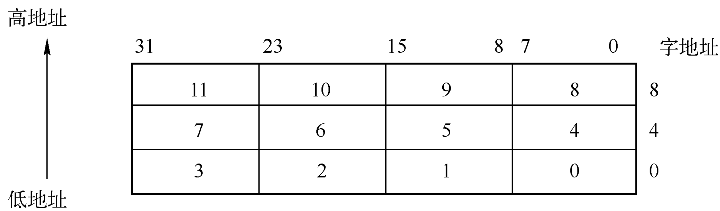


图 1-3　小端存储格式

多数厂家生产的 ARM 核处理器芯片，都能够支持大端和小端两种存储格式，默认为小端存储格式。如果一个基于 ARM 核的处理器芯片将存储器系统配置为其中一种格式（如小端），而实际连接的存储器系统配置为相反的格式（如大端），那么只有以字为单位的指令取指、数据加载和数据保存能够可靠地实现，其他的存储器访问将出现不可预知的结果。

ARM 体系结构通常希望所有的存储器访问能适当地对齐。特别是用于字访问的地址通常应当字对齐，用于半字访问的地址通常应当半字对齐。未按这种方式对齐的存储器访问称为非对齐的存储器访问。

ARM 体系结构完成 I/O 功能的标准方法是使用存储器映射 I/O。这种方法采用特定的存储器地址，当从这些地址加载或向这些地址存储时，它们提供 I/O 功能。典型情况下，从存储器映射 I/O 地址加载用于输入，而向存储器地址 I/O 存储则用于输出。

1.2　ARM 处理器的工作状态和运行模式

ARM 处理器有两种工作状态：当执行 32 位的 ARM 指令时，工作在 ARM 状态；当执

行 16 位的 Thumb 指令时，工作在 Thumb 状态。指令执行过程中处理器可以随时在两种工作状态之间切换，并且工作状态的改变不会影响处理器工作模式和相应寄存器中的内容。ARM 指令集和 Thumb 指令集均有切换处理器状态的指令。

由 ARM 状态进入 Thumb 状态的方法是：当操作数寄存器的状态位（位 0）为 1 时，执行 BX 指令可以使处理器从 ARM 状态切换到 Thumb 状态。此外，如果处理器在 Thumb 状态时发生异常（如 IRQ、FIQ、Undef、Abort 和 SWI 等），则当异常处理返回时，自动切换到 Thumb 状态。

由 Thumb 状态进入 ARM 状态的方法是：当操作数寄存器的状态位（位 0）为 0 时，执行 BX 指令可以使处理器从 Thumb 状态切换到 ARM 状态。此外，当处理器进行异常处理（如 IRQ、FIQ、Undef、Abort 和 SWI 等）时，将程序计数器 PC 指针放入异常模式链接寄存器中，并从异常向量地址开始执行程序，也可以使处理器切换到 ARM 状态。

ARM 处理器支持如表 1-1 所列的 7 种运行模式。

表 1-1　ARM 处理器的运行模式

处理器模式	说　　明
用户模式（usr）	ARM 处理器正常的程序执行状态
系统模式（sys）	运行具有特权的操作系统任务
快中断模式（fiq）	支持高速数据传输或通道处理
管理模式（svc）	操作系统保护模式
数据访问中止模式（abt）	用于虚拟存储器及存储器保护
中断模式（irq）	用于通用的中断处理
未定义指令中止模式（und）	支持硬件协处理器的软件仿真

ARM 处理器的运行模式可以通过软件改变，也可以通过外部中断或异常处理改变。大多数应用程序运行在用户模式下。当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

除了用户模式和系统模式之外的 5 种模式又称为异常模式（Exception Modes）。异常模式通常用于处理中断或异常，以及需要访问被保护的系统资源等情况。为了确保从用户模式进入异常模式的可靠性，每种模式都有一些特定的附加寄存器。

系统模式不能由任何异常进入，它与用户模式有完全相同的寄存器，但不受用户模式的限制。系统模式供需要访问系统资源的操作系统任务使用，且不使用与异常模式有关的附加寄存器，从而保证了当任何异常出现时，操作系统任务的状态都是可靠的。

1.3　寄存器组织

1.3.1　寄存器分类

ARM 处理器共有 37 个 32 位寄存器，分为如下两类：

- 31 个通用寄存器，包括程序计数器 PC。这些寄存器是 32 位的。
- 6 个状态寄存器，也是 32 位的，但只使用了其中的 12 位。

这些寄存器不能被同时访问，具体有哪些寄存器是可访问的，取决于处理器的工作状态和运行模式。寄存器被安排成部分重叠的组，每种处理器模式使用不同的寄存器组。

ARM 状态下的寄存器组织如图 1-4 所示。在任何时候，15 个通用寄存器 R0～R14、程序计数器 PC、程序状态寄存器 CPSR 都是可访问的。

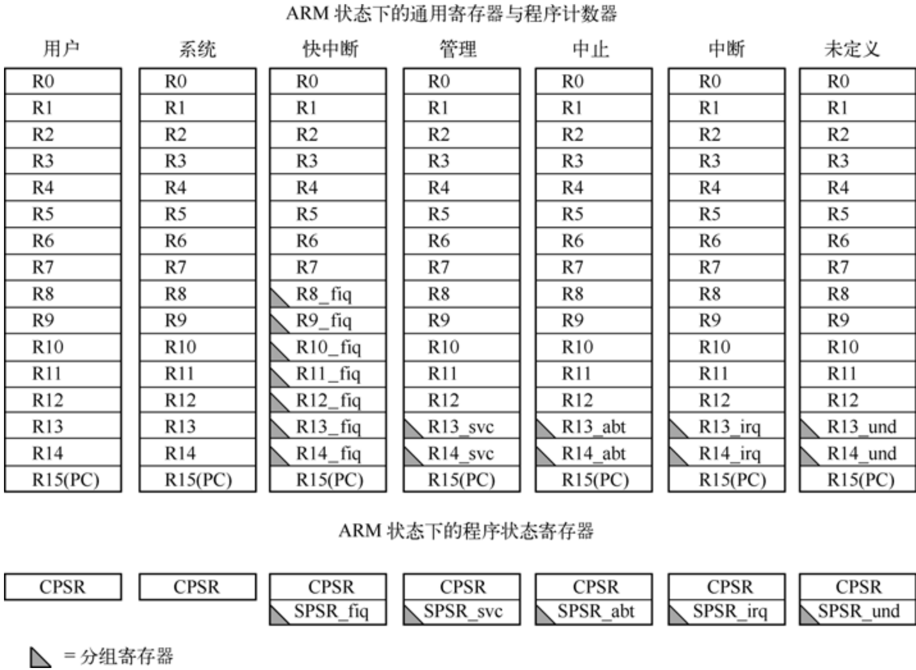


图 1-4　ARM 状态下的寄存器组织

Thumb 状态下的寄存器集是 ARM 状态下寄存器集的一个子集，程序可以直接访问 8 个通用寄存器 R7～R0、程序计数器 PC、堆栈指针 SP、链接寄存器 LR 和 CPSR。每一种特权模式下都有一组 SP、LR 和 SPSR。图 1-5 所示为 Thumb 状态下的寄存器组织。

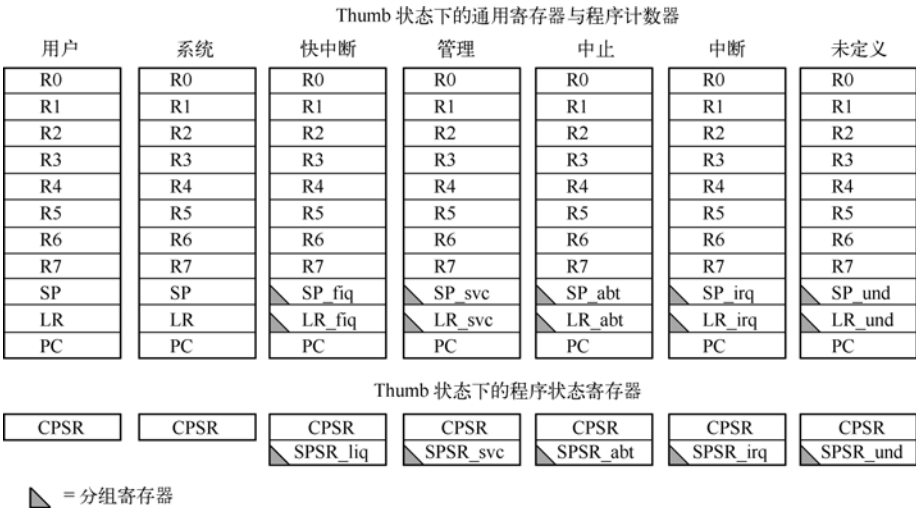


图 1-5　Thumb 状态下的寄存器组织

Thumb 状态下的寄存器组织与 ARM 状态下的寄存器组织的映射关系如图 1-6 所示。其中 Thumb 状态下的 R0～R7 与 ARM 状态下的 R0～R7 相同。Thumb 状态下的 CPSR 和 SPSR 与 ARM 状态下的 CPSR 和 SPSR 相同。Thumb 状态下的 SP 映射到 ARM 状态下的 R13。Thumb 状态下的 LR 映射到 ARM 状态下的 R14。Thumb 状态下的程序计数器 PC 映射到 ARM 状态下的 PC（R15）。

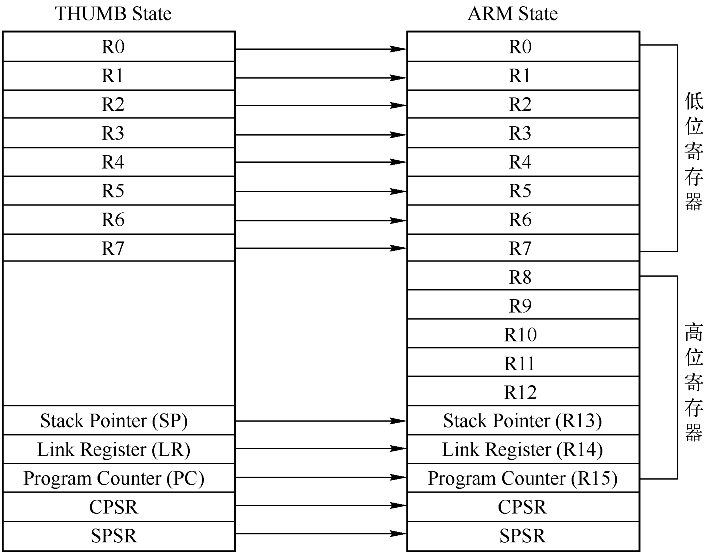


图 1-6　Thumb 状态寄存器映射到 ARM 状态寄存器

在 Thumb 状态下，高位寄存器 R8～R15 并不是标准寄存器集的一部分，但可使用汇编语言程序受限制的访问这些寄存器，将其用作快速的暂存器。使用带特殊变量的 MOV 指令，数据可以在低位寄存器和高位寄存器之间进行传送；高位寄存器的值可以使用 CMP 和 ADD 指令进行比较或加上低位寄存器中的值。

1.3.2　通用寄存器

通用寄存器包括 R0～R15，包括：

- 未分组寄存器 R0～R7。
- 分组寄存器 R8～14。
- 程序计数器 PC（R15）。

1．未分组寄存器 R0～R7

在所有的运行模式下，未分组寄存器都指向同一个物理寄存器，没有特殊用途。因此，在中断或异常处理进行模式切换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在程序设计时应引起注意。

2．分组寄存器 R8～R14

对于分组寄存器，它们每一次所访问的物理寄存器与处理器当前的运行模式有关。若要访问特定的物理寄存器而不依赖于当前的处理器模式，则要使用规定的名字。

对于 R8～R12 来说，每个寄存器对应两个不同的物理寄存器，当使用 fiq 模式时，访问

寄存器 R8_fiq～R12_fiq；当使用除 fiq 模式以外的其他模式时，访问寄存器 R8_usr～R12_usr。

对于 R13、R14 来说，每个寄存器对应 6 个不同的物理寄存器，其中一个由用户模式与系统模式共用，另外 5 个物理寄存器对应于其他 5 种不同的运行模式。

采用以下的记号来区分不同的物理寄存器：

```
R13_<mode>
R14_<mode>
```

其中，mode 为以下几种模式之一：usr、fiq、irq、svc、abt、und。

寄存器 R13 在 ARM 指令中通常用作堆栈指针，简称为 SP。每种运行模式均有自己独立的物理寄存器 R13，在用户应用程序的初始化部分，一般都要将 R13 初始化为指向其运行模式的栈空间，这样，当程序运行进入异常模式时，可以将需要保护的寄存器放入 R13 所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复被保护的寄存器，采用这种方式可以保证异常发生后程序的正常执行。

R14 也称作子程序链接寄存器，简称为 LR。当执行 BL（子程序调用）指令时，R14 中得到 R15（程序计数器 PC）的备份。其他情况下，R14 用作通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器 R14_svc、R14_irq、R14_fiq、R14_abt 和 R14_und 用来保存 R15 的返回值。

在每一种运行模式下，都可用 R14 保存子程序的返回地址。当用 BL 或 BLX 指令调用子程序时，将 PC 的当前值复制给 R14，执行完子程序后，又将 R14 的值复制回 PC，即可完成子程序的调用返回。此外，当异常出现时，相应异常模式的 R14 也可被设置成异常返回地址。异常返回以与子程序返回类似的方法实现，但使用的指令稍有不同。

3．程序计数器 R15

寄存器 R15 通常用作程序计数器 PC。在 ARM 状态下，R15 的位[1:0]为 0，位[31:2]用于保存 PC 的值；在 Thumb 状态下，R15 的位[0]为 0，位[31:1]用于保存 PC 的值。

R15 虽然也可用作通用寄存器，但一般不这么使用，因为对 R15 的使用有一些特殊限制，当违反了这些限制时，程序的执行结果是未知的。

由于 ARM7 体系结构采用了两级流水线技术，对于 ARM 指令集而言，PC 总是指向当前指令的下两条指令的地址，即 PC 的值为当前指令的地址值加 8 个字节。

在 ARM 状态下，任一时刻可以访问以上所讨论的 16 个通用寄存器和一到两个状态寄存器。在非用户模式（特权模式）下，则可访问特定模式分组寄存器。

1.3.3 程序状态寄存器

ARM 处理器包含 1 个当前程序状态寄存器 CPSR 和 5 个程序状态保存寄存器 SPSR。所有运行模式下都可以访问当前程序状态寄存器。每种异常模式都有对应的程序状态保存寄存器，用户模式和系统模式不属于异常，因而没有程序状态保存寄存器。程序状态寄存器的定义如图 1-7 所示，其主要功能包括：

- 保存算术逻辑单元 ALU 中的当前操作信息。
- 控制允许和禁止中断。

● 设置处理器的运行模式。

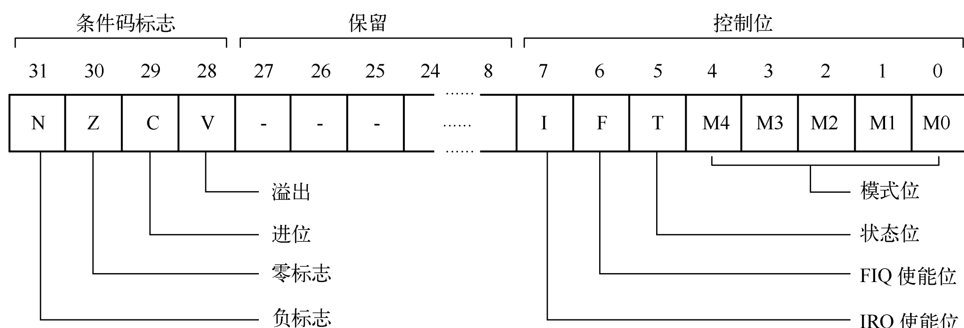


图 1-7　程序状态寄存器

1．条件码标志

图 1-5 中的 N、Z、C、V 均为条件码标志位，它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行。在 ARM 状态下，绝大多数的指令都是有条件执行的。在 Thumb 状态下，仅有分支指令是有条件执行的。

条件码标志位的具体含义如表 1-2 所列。

表 1-2　条件码标志的具体含义

标　志　位	含　　义
N	当用两个补码表示的带符号数进行运算时，N=1 表示运算的结果为负数，N=0 表示运算的结果为正数或零
Z	Z=1 表示运算的结果为零，Z=0 表示运算的结果为非零
C	加法运算（包括比较指令 CMN）：当运算结果产生了进位时（无符号数溢出），C=1，否则 C=0 减法运算（包括比较指令 CMP）：当运算时产生了借位（无符号数溢出），C=0，否则 C=1 对于包含移位操作的非加/减运算指令，C 为移出值的最后一位 对于其他的非加/减运算指令，C 的值通常不改变
V	对于加/减法运算指令，当操作数和运算结果为二进制补码表示的带符号数时，V=1 表示符号位溢出 对于其他的非加/减运算指令，V 的值通常不改变

2．控制位

程序状态寄存器的低 8 位（包括 I、F、T 和 M[4：0]）称为控制位，当发生异常时，这些位可以被改变。如果处理器在特权模式下运行，这些位也可以由程序修改。

中断禁止位 I、F：I=1 表示禁止 IRQ 中断，F=1 表示禁止 FIQ 中断。

标志位 I：该位反映处理器的运行状态，T=0 表示运行于 ARM 状态，T=1 表示运行于 Thumb 状态。

运行模式位 M[4：0]：M0、M1、M2、M3、M4 是运行模式位，它们的状态决定了处理器的运行模式。具体含义如表 1-3 所列。

表 1-3　运行模式位 M[4：0]的具体含义

M[4：0]	处理器模式	可访问的寄存器
0b10000	用户模式	PC、CPSR、R0～R14
0b10001	FIQ 模式	PC、CPSR、 SPSR_fiq、R14_fiq、R8_fiq、 R7～R0
0b10010	IRQ 模式	PC、CPSR、 SPSR_irq、R14_irq、R13_irq、R12～R0

（续）

M[4：0]	处理器模式	可访问的寄存器
0b10011	管理模式	PC、CPSR、 SPSR_svc、R14_svc、R13_svc、R12～R0
0b10111	中止模式	PC、CPSR、 SPSR_abt、R14_abt、R13_abt、 R12～R0
0b11011	未定义模式	PC、CPSR、 SPSR_und、R14_und、R13_und、 R12～R0
0b11111	系统模式	PC、CPSR（ARM v4 及以上版本）、 R14～R0

由表 1-3 可知，并不是所有运行模式位的组合都能定义一个有效的处理器模式，其他组合可能会导致处理器进入一个不可恢复的状态。

程序状态寄存器中的其余位为保留位，保留位用于将来 ARM 版本的扩展。当改变寄存器中的条件码标志位或控制位时，保留位不要被改变，在程序中也不要使用保留位来存储数据。

1.4 异常

当正常的程序执行流程发生暂时停止时，称为异常，例如，为了处理一个外部中断请求，处理器在执行完当前指令后暂停正常流程，而转去执行外部中断异常服务程序。在处理异常之前，当前处理器的状态必须保留，以便异常处理完成之后，可以继续执行原来的程序。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。

ARM 体系结构中的异常与 8 位/16 位体系结构的中断有许多相似之处，但异常与中断的概念并不完全等同。

1.4.1 ARM 体系结构所支持的异常类型

ARM 体系结构所支持的异常及具体含义如表 1-4 所列。异常发生后，ARM 处理器强制从异常类型对应的固定入口地址开始执行程序，这些固定入口地址称为异常向量地址，如表 1-5 所列。

表 1-4　ARM 体系结构所支持的异常类型

异 常 类 型	具 体 含 义
复位	当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行
未定义指令	当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真
软件中断	该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用
指令预取中止	若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常
数据中止	若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常
IRQ（外部中断请求）	当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务
FIQ（快速中断请求）	当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常

表 1-5　异常向量表

异　　常	进 入 模 式	异常向量地址
复位	管理模式	0x00000000
未定义指令	未定义模式	0x00000004
软件中断	管理模式	0x00000008
指令预取中止	中止模式	0x0000000C
数据中止	中止模式	0x00000010
保留	保留	0x00000014
IRQ	IRQ	0x00000018
FIQ	FIQ	0x0000001C

多个异常同时发生时，系统根据固定的优先级，决定异常的处理次序。异常优先级由高到低的排列次序如表 1-6 所列。

表 1-6　异常优先级

优　先　级	异　　常
1（最高）	复位
2	数据中止
3	FIQ
4	IRQ
5	预取指令中止
6（最低）	未定义指令、SWI

1.4.2　各类异常的具体描述

1．复位

当 ARM 处理器的复位信号电平有效时，产生复位异常，处理器立刻停止执行当前指令，在禁止中断的管理模式下，强制从地址 0x00000000 开始执行指令。

2．未定义指令

当 ARM 处理器遇到不能处理的指令时，产生未定义指令异常。采用这种机制，可以通过软件仿真扩展 ARM 或 Thumb 指令集。

无论是在 ARM 状态还是 Thumb 状态，在仿真未定义指令后，处理器执行以下指令返回：

```
MOVS　PC,　R14_und
```

该指令恢复 PC（从 R14_und）和 CPSR（从 SPSR_und）的值，并返回到未定义指令后的下一条指令。

3．软件中断

软件中断指令 SWI 用于进入管理模式，通常用于请求执行特定的管理功能。无论是在 ARM 状态还是 Thumb 状态，软件中断处理程序执行以下指令从 SWI 模式返回：

```
MOV    PC ,    R14_svc
```

该指令恢复 PC（从 R14_svc）和 CPSR（从 SPSR_svc）的值，并返回到 SWI 的下一条指令。

4．指令预取中止

当指令预取访问存储器失败时，存储器系统向 ARM 处理器发出存储器中止（abort）信号，预取的指令被记为无效。但只有当处理器试图执行无效指令时，才会发生预取中止异常。如果指令未被执行，例如在指令流水线中发生了转移，则不会发生预取指令中止。

无论是在 ARM 状态还是 Thumb 状态，确定了中止的原因后，执行以下指令从种植模式返回：

```
MOV    PC, R14_abt, #4
```

该指令恢复 PC（从 R14_abt）和 CPSR（从 SPSR_abt）的值，并重新执行中止的指令。

5．数据中止异常

数据中止发生在数据访问期间。存储器系统发出存储器中止信号，激活中止来响应数据访问（加载或存储），数据被标记为无效。

中止机制使指令分页的虚拟存储器系统能够实现。在这样一个系统中，处理器允许产生仲裁地址。当某一地址的数据无法访问时，存储器管理单元 MMU 通知产生了中止。中止处理程序必须找出中止的原因，使请求的数据可以被访问并重新执行被中止的指令。应用程序不必知道可用存储器的数量，也不必知道其被中止时所处的状态。

无论是在 ARM 状态还是 Thumb 状态，在修复产生数据中止的原因后，执行以下指令从中止模式返回：

```
MOV    PC, R14_abt, #8
```

该指令恢复 PC（从 R14_abt）和 CPSR（从 SPSR_abt）的值，并重新执行中止的指令。

6．中断请求 IRQ 异常

IRQ 异常属于正常的中断请求，可通过对处理器的 nIRQ 端引脚输入低电平产生。IRQ 的优先级低于 FIQ，当程序执行进入 FIQ 异常时，IRQ 可能被屏蔽。

若将 CPSR 的 I 位置为 1，则会禁止 IRQ 中断；若将 CPSR 的 I 位清零，处理器会在指令执行完之前检查 IRQ 的输入。注意只有在特权模式下才能改变 I 位的状态。

不管是在 ARM 状态还是在 Thumb 状态下进入 IRQ 模式，执行以下指令从 IRQ 模式返回：

```
SUBS    PC , R14_irq, #4
```

该指令将寄存器 R14_irq 的值减去 4 后，复制到程序计数器 PC 中，从而实现从异常处理程序返回，同时将 SPSR_mode 寄存器的内容复制到当前程序状态寄存器 CPSR 中。

7．快中断请求 FIQ 异常

FIQ 异常是为了支持数据转移或者通道处理而设计的。在 ARM 状态下，快中断模式有 8 个专用寄存器，用于对寄存器保存的需求，并减小了系统上下文切换的开销。

若将 CPSR 的 F 位置为 1，则会禁止 FIQ 中断，若将 CPSR 的 F 位清零，处理器会在指

令执行时检查 FIQ 输入。注意，只有在特权模式下，才能改变 F 位的状态。

可由外部通过对处理器的 nFIQ 引脚输入低电平产生 FIQ。

不管是在 ARM 状态还是在 Thumb 状态下进入 FIQ 模式，执行以下指令从 FIQ 模式返回：

```
SUBS    PC,R14_fiq,#4
```

该指令将寄存器 R14_fiq 的值减去 4 后，复制到程序计数器 PC 中，从而实现从异常处理程序中的返回，同时将 SPSR_mode 寄存器的内容复制到当前程序状态寄存器 CPSR 中。

1.4.3　对异常的响应和返回

当一个异常出现以后，ARM 处理器会执行以下几步操作进入异常：

1）将下一条指令的地址存入相应链接寄存器 LR，以便程序在处理异常返回时，能从正确的位置重新开始执行。若异常是从 ARM 状态进入，LR 寄存器中保存的是下一条指令的地址（当前 PC+4 或 PC+8，与异常的类型有关）；若异常是从 Thumb 状态进入，则在 LR 寄存器中保存当前 PC 的偏移量。这样，异常处理程序就不需要确定异常是从何种状态进入的。例如：对于软件中断 SWI 异常，指令 MOV PC，R14_svc 总是返回到下一条指令，不管 SWI 是在 ARM 状态执行，还是在 Thumb 状态执行。

2）将 CPSR 复制到相应的 SPSR 中。

3）根据异常类型，设置 CPSR 的运行模式位。

4）强制 PC 从相关异常向量地址取下一条指令执行，从而跳转到相应异常处理程序处。

还可以设置中断禁止位，以阻止其他无法处理的异常嵌套。

如果异常发生时处理器处于 Thumb 状态，则当异常向量地址加载到 PC 时，处理器将自动切换到 ARM 状态。

ARM 处理器对异常的响应过程可以用伪代码描述如下：

```
R14_<Exception mode> = Return Link
SPSR_<Exception mode> = CPSR
CPSR[4:0] = Exception mode number       /*  进入相应异常模式  */
CPSR[5] = 0                             /*  在 ARM 状态下执行  */
if <Exception mode> == Reset or FIQ then
      CPSR[6] = 1                       /*  响应 FIQ 异常时，禁止新的 FIQ 异常  */
      CPSR[7] = 1                       /*  禁止 IRQ 异常  */
PC = Exception Vector Address           /*  根据异常向量进行跳转  */
```

异常处理完毕之后，ARM 处理器会执行以下几步操作从异常返回：

1）将链接寄存器 LR 的值减去相应的偏移量后送到 PC 中。

2）SPSR 复制回 CPSR 中。

3）若在进入异常处理时设置了中断禁止位，要在此清除。

系统运行时异常可能随时发生。为了保证 ARM 处理器在发生异常时不会出于未知状态，应用程序设计中必须确保对异常的处理。通常是在异常向量表中特定位置放置一条跳转指令，当发生异常时，程序计数器 PC 的值被强制设置为对应的异常向量地址，跳转到异常

处理程序。当异常处理完成之后，返回到主程序继续执行。

1.5 ARM 指令集

1.5.1 ARM 指令的功能与格式

ARM 处理器的指令集是加载/存储型的，即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对系统存储器的访问需要通过专门的加载/存储指令来完成。

ARM 处理器的指令集可以分为跳转指令、数据处理指令、程序状态寄存器处理指令、加载/存储指令、协处理器指令和异常产生指令六大类。具体的指令及功能如表 1-7 所列（表中指令为基本 ARM 指令，不包括派生的 ARM 指令）。

表 1-7　ARM 指令及功能描述

指令助记符	指令功能描述
ADC	带进位加法指令
ADD	加法指令
AND	逻辑与指令
B	跳转指令
BIC	位清零指令
BL	带返回的跳转指令
BLX	带返回和状态切换的跳转指令
BX	带状态切换的跳转指令
CDP	协处理器数据操作指令
CMN	比较反值指令
CMP	比较指令
EOR	异或指令
LDC	存储器到协处理器的数据传输指令
LDM	加载多个寄存器指令
LDR	存储器到寄存器的数据传输指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令
MLA	乘加运算指令
MOV	数据传送指令
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
MUL	32 位乘法指令
MLA	32 位乘加指令
MVN	数据取反传送指令
ORR	逻辑或指令
RSB	逆向减法指令

（续）

指令助记符	指令功能描述
RSC	带借位的逆向减法指令
SBC	带借位减法指令
STC	协处理器寄存器写入存储器指令
STM	批量内存字写入指令
STR	寄存器到存储器的数据传输指令
SUB	减法指令
SWI	软件中断指令
SWP	交换指令
TEQ	相等测试指令
TST	位测试指令

ARM 指令的基本格式如下：

<opcode> {<cond>} {S} <Rd>，<Rn>，{<oprand2>}

其中，<>内是必需项，{}内是可选项。

opcode　指令助记符，如 LDR、STR 等。

cond　执行条件，如 EQ、NE 等。

S　是否影响 CPSR 寄存器的值，书写时影响，否则不影响。

Rd　目标寄存器。

Rn　第一个操作数寄存器。

oprand2　第二个操作数。

1.5.2　指令的条件码

当处理器工作在 ARM 状态时，几乎所有的指令都会根据 CPSR 中条件码的状态和指令的条件域有条件的执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。

每一条 ARM 指令包含 4 位条件码，位于指令的最高 4 位[31:28]。条件码共有 16 种，如表 1-8 所列。每种条件码用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令 B 可以加上扩展名 EQ 变为 BEQ，表示“相等则跳转”，即当 CPSR 中的 Z 标志置位时发生跳转。

表 1-8　指令的条件码

条　件　码	助记符后缀	标　　志	含　　义
0000	EQ	Z 置位	相等
0001	NE	Z 清零	不相等
0010	CS	C 置位	无符号数大于或等于
0011	CC	C 清零	无符号数小于
0100	MI	N 置位	负数
0101	PL	N 清零	正数或零

（续）

条　件　码	助记符后缀	标　　志	含　　义
0110	VS	V 置位	溢出
0111	VC	V 清零	未溢出
1000	HI	C 置位 Z 清零	无符号数大于
1001	LS	C 清零 Z 置位	无符号数小于或等于
1010	GE	N 等于 V	带符号数大于或等于
1011	LT	N 不等于 V	带符号数小于
1100	GT	Z 清零且（N 等于 V）	带符号数大于
1101	LE	Z 置位或（N 不等于 V）	带符号数小于或等于
1110	AL	忽略	无条件执行
1111	保留	未使用	未使用

在 16 种条件标志码中，只有 15 种可以使用，第 16 种（1111）为系统保留，暂时不能使用。

1.6　ARM 指令的寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。ARM 指令系统支持如下 9 种常见的寻址方式。

1.6.1　寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，指令执行时直接取出寄存器的内容来操作。例如以下指令：

```
ADD  R0，R1，R2                ；R0←R1＋R2
```

这条指令将寄存器 R1 和 R2 的内容相加，结果存放在寄存器 R0 中。操作数的顺序为：第一个是结果寄存器，然后是第一操作数寄存器，最后是第二操作数寄存器。

1.6.2　立即寻址

立即寻址又称立即数寻址，这是一种特殊的寻址方式。操作数本身就在指令中给出，只要取出指令也就取到了操作数，这个操作数被称为为立即数，对应的寻址方式也就称为立即寻址。例如以下指令：

```
ADD  R0，R0，#1                ；R0←R0＋1
ADD  R0，R0，#0x3f             ；R0←R0＋0x3f
```

在以上两条指令中，第二个操作数即为立即数，要求以“＃”为前缀，对于以十六进制表示的立即数，还要求在“＃”后加上“0x”或“&”。当立即数大于第二操作数的表示范围时，通常用以下伪指令实现：

```
LDR   R0, =#0xffff0000
```

1.6.3 寄存器移位寻址

这种寻址方式是 ARM 指令集特有的，当第二个操作数为寄存器时，可以加入移位选项对它进行移位操作。可以采取的移位操作如下：

- LSL：逻辑左移 n 位（1<=n<=32），寄存器中字的低端空出位补 0。
- LSR：逻辑右移 n 位（1<=n<=32），寄存器中字的高端空出位补 0。
- ASR：算术右移 n 位（1<=n<=32），算术移位的对象是带符号数，在移位过程中必须保持操作数的符号不变。若操作数为正数，则字的高端空出位补 0；若操作数为负数，则字的高端空出位补 1。
- ROR：循环右移 n 位（1<=n<=32），从字的最低端移出的位填入到字的高端空出位。
- RRX：带扩展的循环右移 1 位。按操作数所指定的数量向右循环移位，空位（位[31]）用原来的进位标志 C 填充。

各种移位操作如图 1-8 所示。

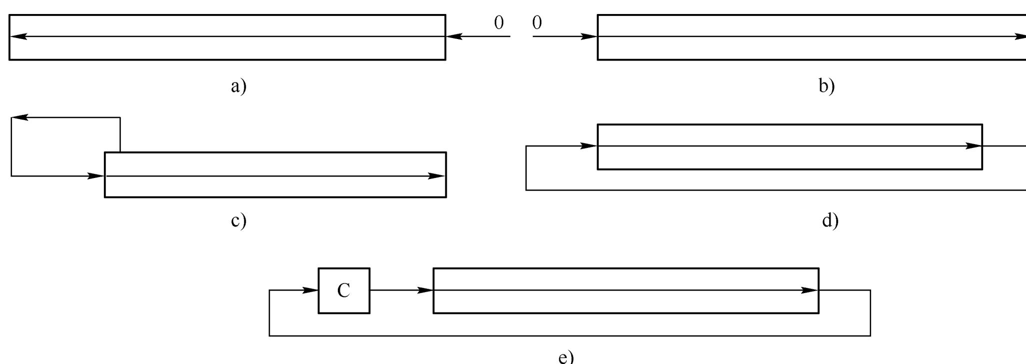


图 1-8　移位操作过程

a) LSL 移位操作　b) LSR 移位操作　c) ASR 移位操作　d) ROR 移位操作　e) RRX 移位操作

例如以下指令：

```
MOV    R3, R2, LSL #2         ；R2 的值逻辑左移 2 位，结果放入 R3 中
MOV    R3, R2, LSR #2         ；R2 的值逻辑右移 2 位，结果放入 R3 中
MOV    R3, R2, RRX            ；R2 的值带扩展循环右移 1 位，结果放入 R3 中
ADD    R3, R2, R1, ASR #2     ；R1 的值算数右移 2 位，再与 R2 的值相加，结果放入 R3 中
SUB    R3, R2, R1, ROR #2     ；R1 的值循环右移 2 位，再与 R2 的值相减，结果放入 R3 中
```

1.6.4 寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如以下指令：

```
LDR  R0，[R1]          ；R0←[R1]
STR  R3，[R2]          ；[R2]←R3
```

第一条指令将以 R1 的值为地址的存储器中的数据传送到 R0 中。

第二条指令将 R3 的值传送到以 R2 的值为地址的存储器中。

1.6.5 基址寻址

基址寻址就是将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，形成第一个操作数的有效地址。基址寻址方式常用于访问基地址附近的存储单元，如查表、数组等操作。采用基址寻址方式的指令常见有以下几种形式：

```
LDR R0，[R1，#4]          ；R0←[R1＋4]
LDR R0，[R1，#4]!         ；R0←[R1＋4]、R1←R1＋4
LDR R0，[R1]，#4          ；R0←[R1]、R1←R1＋4
LDR R0，[R1，R2]          ；R0←[R1＋R2]
```

第一条指令，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从该地址取得操作数存入寄存器 R0 中。

第二条指令，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从该地址取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

第三条指令，以寄存器 R1 的内容作为操作数的有效地址，从该地址取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

第四条指令，将寄存器 R1 的内容加上寄存器 R2 的内容形成操作数的有效地址，从该地址取得操作数存入寄存器 R0 中。

1.6.6 相对寻址

相对寻址是基址寻址的一种变通，以程序计数器 PC 的当前值作为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段中的跳转指令采用了相对寻址方式：

```
        BL      NEXT                ；调用子程序 NEXT
        BEQ     LOOP                ；条件跳转到 LOOP 标号处
        … …
LOOP    MOV     R6, #1
        … …
NEXT    ……                          ；子程序 NEXT
```

1.6.7 多寄存器寻址

多寄存器寻址是 ARM 处理器特有的一种寻址方式。采用多寄存器寻址，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多 16 个通用寄存器的值。例如以下指令：

```
LDMIA R0，{R1，R2，R3，R4}      ；R1←[R0]
                               ；R2←[R0＋4]
                               ；R3←[R0＋8]
                               ；R4←[R0＋12]
```

该指令的扩展名 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加。因此，指令可将连续存储单元的值传送到 R1～R4。多个连续的寄存器可以用短划线“-”连接，不连续的寄存器用逗号“，”隔开。例如上述指令也可以写成：

```
LDMIA R0，{R1-R4}
或
LDMIA R0，{R1-R3，R4}
```

1.6.8 堆栈寻址

堆栈寻址是多寄存器寻址的一种特殊形式。堆栈是一种按特定顺序进行存取的存储区，操作堆栈的顺序可归结为“后进先出”或“先进后出”。堆栈寻址使用一个称作堆栈指针的专用寄存器指向一块存储区域，指针所指向的存储单元称为栈顶。

根据存储器地址增长方式，堆栈可分为如下两种形式：

- 向上生长：即向高地址方向生长，称为递增堆栈。
- 向下生长：即向低地址方向生长，称为递减堆栈。

当堆栈指针指向最后压入堆栈的数据时，称为满堆栈（Full Stack）；当堆栈指针指向下一个将要放入数据的空位置时，称为空堆栈（Empty Stack）。这样就有 4 种类型的堆栈工作方式，ARM 微处理器支持这 4 种类型的堆栈工作方式，即：

- 满递增堆栈：堆栈指针指向最后压入的数据，且由低地址向高地址生长。
- 满递减堆栈：堆栈指针指向最后压入的数据，且由高地址向低地址生长。
- 空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生长。
- 空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生长。

堆栈寻址指令举例如下：

```
STMFD   SP!，{R1-R7，LR}        ；将 R1～R7、LR 的内容入栈，满递减堆栈
LDMFD   SP!，{R1-R7，LR}        ；数据出栈，放入 R1～R7、LR，满递减堆栈
```

1.6.9 块拷贝寻址

块拷贝寻址也是多寄存器寻址的一种特殊形式，通常用于内存拷贝。这种寻址方式利用多寄存器传送指令将一块数据从存储器的某个位置拷贝到另一个位置。

块拷贝寻址指令举例如下：

```
STMIA   R0!, {R1-R7}     ；将 R1～R7 的内容保存到存储器中，存储器指针在保存第
                          一个值之后向上增长
STMIB   R0!, {R1-R7}     ；将 R1～R7 的内容保存到存储器中，存储器指针在保存第
                          一个值之前向上增长
STMDA   R0!, {R1-R7}     ；将 R1～R7 的内容保存到存储器中，存储器指针在保存第
                          一个值之后向下增长
STMDB   R0!, {R1-R7}     ；将 R1～R7 的数据保存到存储器中，存储器指针在保存第
                          一个值之前向下增长
```

1.6.10 ARM 伪指令

ARM 伪指令不是 ARM 指令集中的指令，而是编译器设置的一种为方便编程的“假”指令。伪指令可以像其他 ARM 指令一样使用，但在编译过程中，这些指令将被等效的 ARM 指令所取代。

1．ADR（小范围地址读取）

ADR 伪指令将基于当前 PC 相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。

指令格式：

```
ADR{条件} reg, expr
```

式中，reg 为加载的目的寄存器；expr 为相对偏移表达式，非字对齐时取值范围为–255～255B。字对齐时取值范围为–1020～1020B。

指令示例：

```
start:  MOV R0, #10
        ADR R4, start    ; 相当于 SUB R4, PC,#0x0C
```

2．ADRL（中范围地址读取）

ADRL 伪指令与 ADR 类似，但相对偏移量的取值范围有所不同，非字对齐地址时取值范围为 64KB。字对齐地址时取值范围为 256KB。

3．LDR（大范围地址读取）

LDR 伪指令将 32 位常量或一个地址加载到指定寄存器。

指令格式：

```
LDR{条件}  reg, =expr/label_expr
```

式中，reg 为加载的目的寄存器；expr 为 32 位立即数；label_expr 为基于 PC 的地址表达式或外部表达式。

指令示例：

```
LDR  R0,=0x12345          ；加载 32 位立即数 0x12345 到寄存器 R0
LDR  R0,=DATA_BUF+60      ；加载 DATA_BUF 地址+60 到寄存器 R0
```

4．NOP（空操作）

NOP 是空操作伪指令。NOP 伪指令在汇编时将会被替换为 ARM 指令中的空操作，例如：可能是 MOV R0,R0 指令等。另外，NOP 指令还可以用于软件延时。

指令格式：

```
NOP
```

1.7 Thumb 指令集

为了兼容数据总线宽度为 16 位的应用系统，ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的

一个子集，允许指令编码为 16 位的长度。与等价的 32 位代码相比较，Thumb 指令集在保留 32 代码优势的同时，大大节省了系统的存储空间。

Thumb 指令都有对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型，在编写应用程序时，只要遵循一定的调用规则，Thumb 子程序和 ARM 子程序就可以互相调用。

当处理器在执行 ARM 程序段时，处理器处于 ARM 工作状态，当处理器在执行 Thumb 程序段时，处理器处于 Thumb 工作状态。

与 ARM 指令集相比较，Thumb 指令集中数据处理指令的操作数仍然是 32 位，指令地址也为 32 位，但是，Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性，如大多数的 Thumb 指令是无条件执行的，而几乎所有的 ARM 指令都是有条件执行的。大多数的 Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于 Thumb 指令的长度为 16 位，即只用 ARM 指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需 Thumb 指令的条数较 ARM 指令多。

一般情况下，指令与 ARM 指令的时间效率和空间效率的关系为：

- Thumb 代码所需的存储器空间为 ARM 代码的 60%～70%。
- Thumb 代码使用的指令条数比 ARM 代码多 30%～40%。
- 若采用 32 位的存储器，ARM 代码比 Thumb 代码快 40%。
- 若采用 16 位的存储器，Thumb 代码比 ARM 代码快 40%～50%。
- 与 ARM 代码相比，使用 Thumb 代码将使存储器的功耗降低 30%。

虽然 Thumb 指令集与 ARM 指令集各有优点，若对系统的性能有较高要求，则应采用 32 位的存储器系统和 ARM 指令集；若对系统的成本和功耗有较高要求，则应采用 16 位的存储器系统和 Thumb 指令集。当然最好是两者结合使用，充分发挥其各自优点，以便取得最好效果。

1.8 ARM 汇编语言编程

采用汇编语言进行应用程序设计，可以直接操作系统的底层硬件接口，而采用 C 或 C++等高级语言编程，则可提高工作效率。通常 ARM 编译器都支持汇编语言和 C 语言以及二者混合编程。

1.8.1 ARM 汇编语言规范

在 ARM 汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定名称。段可以分为代码段和数据段，代码段为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时，最终形成一个可执行的映像文件。

可执行映像文件通常由以下几部分构成：

- 一个或多个代码段，代码段的属性为只读。
- 零个或多个包含初始化数据的数据段，数据段的属性为可读写。
- 零个或多个不包含初始化数据的数据段，数据段的属性为可读写。

链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此，源程序中段之间的相对位置与可执行映像文件中段的相对位置可能不相同。

ARM 汇编语言的语句格式如下：

{标号}　指令或伪指令　操作数　　　{；注释}

其中：标号必须顶格书写，并且不能包含任何空白字符。

指令或伪指令不能顶格书写，前面至少应有一个空白。

位于“；”以后的部分为注释，可以顶格书写，注释不参与汇编。

汇编器对大小写字符敏感，源程序中，标号可以采用大小写混合方式，而指令、伪指令、寄存器名称则只能采用大写或小写字符，不能混合使用大小写。

1.8.2　汇编伪指令

在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，它们没有相对应的操作码，通常称这些特殊指令助记符为伪指令，其所完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作的，它们仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成了。

除了前面介绍过的 ARM 伪指令之外，还有其他一些汇编伪指令：符号定义伪指令、数据定义伪指令、段定义伪指令、模块控制伪指令、汇编控制伪指令、宏处理伪指令等。需要注意的是，除了前面介绍的几条 ARM 伪指令外，其他伪指令依赖于编译器，也就是说，不同编译器所支持的伪指令有所不同，这就使得在不同编译器环境下编写的汇编语言程序不尽相同。在进行具体程序设计之前，应该先对所采用编译器功能有一定的了解。

下面是几条常用的汇编伪指令。

1．ARM，CODE32

ARM 和 CODE32 伪指令用于标示其后的指令为 32 位 ARM 指令。

2．THUMB，CODE16

THUMB 和 CODE16 伪指令用于标示其后的指令为 16 位 THUMB 指令。

3．EQU

EQU 伪指令用于为程序模块中的常量、标号等赋值。

4．DCB，DCD

DCB 和 DCD 伪指令分别用于分配一片连续的 8 位字节和 32 位字存储单元，并用伪指令中指定的表达式初始化。

5．END

END 伪指令用于标示一个汇编语言源程序文件的结束。

1.8.3　程序设计举例

例 1-1　简单运算程序，使用 ADD，SUB，LSL，LSR，AND，ORR 等 ARM 指令完成基本数学和逻辑运算。

```
X EQU 45                        ；定义变量 X，并赋值为 45
Y EQU 64                        ；定义变量 Y，并赋值为 64
```

```
Z EQU 87                                 ; 定义变量 Z，并赋值为 87
 NAME ex2
        PUBLIC __iar_program_start
 SECTION `.text`:CODE:NOROOT(2)
 CODE32
__iar_program_start
main:  MOV    R0, #X                     ; R0=X
       MOV    R0, R0, LSL #2             ; R0=X*4
       MOV    R1, #Y                     ; R1=Y
       ADD    R2, R0, R1, LSR #1         ; R2=X*4+Y/2
       MOV    SP, #0X40000000
       STR    R2, [SP]                   ; X*4+Y/2 存入 0x40000000 地址处
       MOV    R0, #Z                     ; R0=Z
       AND    R0, R0, #0XFF              ; 取 R0 的低八位
       MOV    R1, #Y                     ; R1=Y
       ADD    R2, R0, R1, LSR #1         ; R2=Z+Y/2
       LDR    R0, [SP]                   ; R0=X*4+Y/2
       MOV    R1, #0X01
       ORR    R0, R0, R1                 ; R0=R0+1
       MOV    R1, R2                     ; R1=Z+Y/2
       ADD    R2, R0, R1, LSR #1         ; R2=(Z+Y/2)/2+(X*4+Y/2)
STOP:  B      STOP                       ; 停止
       END                               ; 程序结束
```

例 1-2 简单分支程序，利用跳转指令根据不同条件调用不同的子程序。

```
NUM EQU 8                                ; 定义用于条件判断的无符号参数
      NAME EX3
      PUBLIC __iar_program_start
      SECTION `.text`:CODE:NOROOT(2)
      CODE32
__iar_program_start
main:     MOV     R0, #7                 ; R0,R1,R2 装入初值
          MOV     R1, #3
          MOV     R2, #2
          CMP     R0, #NUM               ; 比较 R0 与 NUM 的大小
          BHI     TOSUB                  ; 若 R0>NUM，则调用减法子程序
          BL      DOADD                  ; 否则调用加法子程序
          B       STOP                   ; 转移至程序结束点
TOSUB:    BL      DOSUB                  ; 调用减法子程序
STOP:     B       STOP                   ; 程序结束
DOADD:    ADD     R0, R1, R2             ; 加法子程序
          MOV     PC, LR                 ; 返回
DOSUB:    SUB     R0, R1, R2             ; 减法子程序
          MOV     PC,LR                  ; 返回
          END                            ; 程序结束
```

例 1-3 已知循环次数的单循环程序，实现 s=1+2*3+3*4+…+N(N+1)当 N=10 截止。

```
          PRESERVE8                         ；用 Keil MDK 编译时，这条语句一定要加
                CODE32
          AREA    EX1,CODE,READONLY
                ENTRY
MAIN                                        ；程序代码开始标志
                MOV R0,#1                   ；R0:S,循环初值为 1
                MOV R1,#2                   ；R1:N
REPEAT ADD R2,R1,#1                         ；R2:N+1 程序循环体
                MUL R3,R1,R2                ；R3:N(N+1)
                ADD R0,R0,R3
                ADD R1,R1,#1                ；循环修改
                CMP R1,#10                  ；循环修改
                BLE REPEAT
STOP            B STOP
          END
```

1.8.4 汇编语言与 C/C++混合编程

基于 ARM 的芯片多数为复杂片上系统，对于这种复杂系统，如果全部采用汇编语言编程，其工作量将非常大，而且效率低下，同时也不利于系统升级或应用软件移植。事实上，ARM 体系结构支持 C/C++以及与汇编语言的混合编程。

汇编语言与 C/C++混合编程通常有几种方式：

- 在 C/C++代码中嵌入汇编指令。
- 在汇编语言程序和 C/C++程序之间利用变量进行互访。
- 汇编语言程序与 C/C++程序相互调用。

可以将汇编语言程序和 C/C++程序中需要共享的变量或函数分别声明为全局变量或全局函数，然后在其他程序文件中进行访问和调用。但是从好的编程风格来说，应尽量减少全局变量和全局函数的使用。

在混合编程中，必须遵守一定的调用规则，如物理寄存器的使用、参数的传递等。ARM 公司为此专门制定了一个 ATPCS 标准，但该标准十分繁琐，使用起来不太方便。

实际上使用较多的方法是，初始化部分采用汇编语言完成，其他主要编程工作都由 C/C++完成。系统在完成硬件底层初始化工作之后，跳转到 C/C++程序中运行。汇编语言程序与 C/C++程序之间不要进行太多参数传递和频繁的相互调用，这将使整个程序结构显得简单，易于理解和维护。

1.9 启动代码

基于 ARM 的芯片多数为复杂片上系统，由于 C 语言具有模块性和可移植性的特点，大部分基于 ARM 的应用系统程序都采用 C 语言编写。但是启动系统进入 C 语言的 main 函数之前，需要有一段启动代码来完成对底层硬件的初始化工作，启动代码一般采用汇编语言与 C 语言混合编程，实现异常向量表定义、堆栈初始化、系统变量初始化、中断系统

初始化等操作。

ARM 公司只设计内核而不生产芯片，其他厂商购买内核授权后加入自己的集成外围功能，然后生产出各具特色的 ARM 核处理器芯片，这样促进了基于 ARM 核处理器的芯片多元化，但也使得各种芯片的启动代码差别很大。

下面以 NXP 公司推出的 ARM 核处理器 LPC2138 为例进行介绍。整个启动代码包括汇编文件 Startup.s、C 文件 Target.c、头文件 Config.h、Target.h 等。

1.9.1 Startup.s 文件

Startup.s 文件是 ARM 处理器上电复位后最先运行的启动代码，其中包括各种运行模式下堆栈长度定义、常量定义、外部标号声明、异常向量表、各种模式下系统堆栈的初始化等。

下面列出了 IAR EWARM 软件包提供的 Startup.s 文件，实际应用中可以根据具体系统要求适当修改，以满足不同场合的需要。如果用户程序中没有包含 Startup.s 文件，IAR EWARM 编译器将自动包含一个默认的 Startup 代码。

```
; ----------------------------------------------------------------------
        MODULE    ?cstartup
        ; 存储器段声明
        SECTION IRQ_STACK:DATA:NOROOT(3)
        SECTION FIQ_STACK:DATA:NOROOT(3)
        SECTION CSTACK:DATA:NOROOT(3)
; ----------------------------------------------------------------------
; DLIB 库中包含本文件模块，可以采用自定义启动模块进行替换，自定义模块中应
  定义 PUBLIC 符号__iar_program_start 或其他用户起始符号。替换时只要将用户
  自定义启动模块文件加入到项目中即可
; ----------------------------------------------------------------------
        SECTION .intvec:CODE:NOROOT(2)
        PUBLIC    __vector
        PUBLIC    __iar_program_start
        EXTERN    Undefined_Handler
        EXTERN    SWI_Handler
        EXTERN    Prefetch_Handler
        EXTERN    Abort_Handler
        EXTERN    IRQ_Handler
        EXTERN    FIQ_Handler

        ARM

__iar_init$$done:                ; 完成复制初始化之前不需要向量表

__vector:
        ; 所有默认异常句柄（复位除外）都按 weak 符号定义。应用程序中定义的句柄
        ; 具有更高的优先级。
        LDR       PC,Reset_Addr              ; 复位
```

```
        LDR     PC,Undefined_Addr           ; 未定义指令
        LDR     PC,SWI_Addr                 ; 软件中断(SWI/SVC)
        LDR     PC,Prefetch_Addr            ; 预取中止
        LDR     PC,Abort_Addr               ; 数据中止
        DCD     0                           ; 保留
        LDR     PC,IRQ_Addr                 ; IRQ
        LDR     PC,FIQ_Addr                 ; FIQ

    DATA

Reset_Addr:         DCD     __iar_program_start
Undefined_Addr:     DCD     Undefined_Handler
SWI_Addr:           DCD     SWI_Handler
Prefetch_Addr:      DCD     Prefetch_Handler
Abort_Addr:         DCD     Abort_Handler
IRQ_Addr:           DCD     IRQ_Handler
FIQ_Addr:           DCD     FIQ_Handler

; ---------------------------------------------------------------------
; ?cstartup – 系统底层初始化代码。复位后从此处开始运行。
; CPU 为 ARM 状态、管理模式、禁止中断
; ---------------------------------------------------------------------
        SECTION .text:CODE:NOROOT(2)
        EXTERN   __cmain
        REQUIRE __vector
        EXTWEAK __iar_init_core
        EXTWEAK __iar_init_vfp

        ARM
__iar_program_start:
?cstartup:
; --------------------------------------------------------------------
; 在这里加入设置堆栈指针之前所需要的初始化代码
;
; 初始化堆栈指针
; 以下方式适用于任何异常堆栈：FIQ, IRQ, SVC, ABT, UND, SYS
; 用户模式使用与系统模式相同的堆栈
; 堆栈段必须在链接器命令文件中定义，并且已经在上面声明
; ---------------------------------------------------------------------
; 模式，对应于 CPSR 寄存器的 0-5 位

MODE_MSK     DEFINE      0x1F      ; 用于 CPSR 模式位的位屏蔽

USR_MODE     DEFINE    0x10        ; 用户模式
FIQ_MODE     DEFINE    0x11        ; 快中断请求模式
IRQ_MODE     DEFINE    0x12        ; 中断请求模式
```

```
SVC_MODE    DEFINE    0x13        ; 管理模式
ABT_MODE    DEFINE    0x17        ; 中止模式
UND_MODE    DEFINE    0x1B        ; 为定义指令模式
SYS_MODE    DEFINE    0x1F        ; 系统模式

        MRS     r0, cpsr                    ; 初始 PSR 值

        ;设置 IRQ 中断堆栈指针
        BIC     r0, r0, #MODE_MSK           ; 清零模式位
        ORR     r0, r0, #IRQ_MODE           ; 设置 IRQ 模式位
        MSR     cpsr_c, r0                  ; 改变模式
        LDR     sp, =SFE(IRQ_STACK)         ; IRQ_STACK 堆栈结束
        BIC     sp,sp,#0x7                  ; 保证 SP 为 8 字节对齐

        ; 设置 FIQ 中断堆栈指针
        BIC     r0, r0, #MODE_MSK           ; 清零模式位
        ORR     r0, r0, #FIQ_MODE           ; 设置 FIR 模式位
        MSR     cpsr_c, r0                  ; 改变模式
        LDR     sp, =SFE(FIQ_STACK)         ; FIQ_STACK 堆栈结束
        BIC     sp,sp,#0x7                  ; 保证 SP 为 8 字节对齐

        ; 设置一般堆栈指针
        BIC     r0 ,r0, #MODE_MSK           ; 清零模式位
        ORR     r0 ,r0, #SYS_MODE           ; 设置系统模式位
        MSR     cpsr_c, r0                  ; 改变模式
        LDR     sp, =SFE(CSTACK)            ; CSTACK 堆栈结束
        BIC     sp,sp,#0x7                  ; 保证 SP 为 8 字节对齐

        BL      __iar_init_core
        BL      __iar_init_vfp

        ; 在这里加入更多初始化代码

        B       __cmain
        END
```

1.9.2 Target.c 文件

Target.c 文件包含必要的 IRQ 和 FIQ 的异常处理函数，以及与硬件目标板初始化相关的代码，如 Remap 设置、系统时钟设置、存储器加速模块设置、向量中断控制器初始化等。

Target.c 文件如下：

```
#define IN_TARGET
#include "config.h"

/*********************************************************************************
** 函数名称: IRQ_Exception
```

```
** 功能描述: 中断异常处理程序，用户根据需要自己改变程序
********************************************************************************/
__irq void IRQ_Handler (void)
{
   void (*interrupt_function)();
   unsigned int vector;
   vector = VICVectAddr;                    // 取得中断向量地址
   interrupt_function = (void(*)())vector;
   (*interrupt_function)();                 // 跳转到中断函数
}

/********************************************************************************
** 函数名称: FIQ_Exception
** 功能描述: 快速中断异常处理程序，用户根据需要自己改变程序
********************************************************************************/
        void FIQ_Exception(void)
{
     while(1);                              // 这一句替换为自己的代码
}

/********************************************************************************
** 函数名称: TargetResetInit
** 功能描述: 调用 main 函数之前目标板初始化代码，根据需要改变，不能删除
********************************************************************************/
void TargetResetInit(void)
{
     /* 添加自己的代码 */
}

/********************************************************************************
** 以下为一些与系统相关的库函数的实现，用户可以根据自己的要求修改
********************************************************************************
… …  //略
```

1.9.3　Target.h 文件

Target.h 文件包含需要用到的各种外部函数定义，文件如下：

```
#ifndef IN_TARGET
extern void IRQ_Exception(void);
extern void FIQ_Exception(void);
extern void TargetInit(void);
extern void TargetResetInit(void);
#endif
```

1.9.4 Config.h 文件

Config.h 文件包含一些数据类型定义和系统时钟定义，文件如下：

```
#ifndef __CONFIG_H
#define __CONFIG_H

//定义 BOOL 变量
#ifndef TRUE
#define TRUE    1
#endif

#ifndef FALSE
#define FALSE 0
#endif

//定义变量类型
typedef unsigned    char     uint8;          // 定义无符号 8 位整型变量
typedef signed      char     int8;           // 定义有符号 8 位整型变量
typedef unsigned    short    uint16;         // 定义无符号 16 位整型变量
typedef signed      short    int16;          // 定义有符号 16 位整型变量
typedef unsigned    int      uint32;         // 定义无符号 32 位整型变量
typedef signed      int      int32           // 定义有符号 32 位整型变量
typedef float                fp32;           // 定义单精度浮点数（32 位长度）
typedef double               fp64;           // 定义双精度浮点数（64 位长度）

/**********************************/
/*        LPC2138 的寄存器定义        */
/**********************************/
#include    <iolpc2138.h>

// 系统时钟设置，Fosc、Fcclk、Fcco、Fpclk 必须定义，根据需要改动
#define Fosc        12000000        //晶振频率，应当与实际一致
#define Fcclk       (Fosc * 5)      //系统频率，必须为 Fosc 的整数倍（1～32），且<=60MHz
#define Fcco        (Fcclk * 4)     //CCO 频率，必须为 Fcclk 的 2、4、8、16 倍
#define Fpclk       (Fcclk / 4) * 1 //VPB 时钟频率，只能为（Fcclk / 4）的 1、2、4 倍

#include    "target.h"              //这一句不能删除
#include    <stdio.h>

#endif
```

复习思考题

1．分别列出 ARM 体系结构的数据存储大端格式（Big Endian）和小端格式（Little

Endian）。

2．ARM 处理器有几种工作状态？它们各有什么特点？

3．分别列出 ARM 处理器运行模式。

4．ARM 处理器共有多少个 32 位寄存器？其中 LR 和 PC 分别使用哪个寄存器？

5．R13 寄存器的通用功能是什么？

6．程序状态寄存器 CPSR 有哪些条件码标志位？它们的具体含义是什么？

7．程序状态寄存器 CPSR 的中断禁止位 I、F 的具体含义是什么？

8．列出 ARM 处理器的异常向量表。

9．列出 ARM 处理器在异常出现后将执行的操作步骤。

10．ARM 指令系统支持哪几种寻址方式？

11．ARM 指令集与 Thumb 指令集有什么不同？

第 2 章　Proteus for ARM7 虚拟仿真

2.1　Proteus for ARM 简介

Proteus 是英国 Labcenter Electronics 公司开发的电路分析与实物仿真软件，它运行于 Windows 操作系统上，是一个基于 ProSPICE 混合模型仿真器的、完整的嵌入式系统软、硬件设计仿真平台。Proteus 支持各种主流微处理器及其应用系统的仿真，例如 ARM7（LPC21xx）、8051/52 系列、AVR 系列、PIC10/12/16/18/24 系列、HC11 系列等，并支持多种外围芯片。Proteus 实现了 CPU 仿真和 SPICE 电路仿真的结合，具有模拟电路仿真、数字电路仿真、CPU 及其外围电路组成的系统仿真、RS232 动态仿真、I^2C 调试器、SPI 调试器、USB 协议分析仪、以太网动态仿真、键盘和 LCD 仿真等功能。提供软件调试，具有全速、单步、设置断点等调试功能，同时可以观察各种变量、寄存器等的当前状态。支持多家第三方编译软件，如 Keil C51、Keil MDK for ARM、IAR EWARM、PIC MPLAB 等。另外还提供各种虚拟仪器，如示波器、逻辑分析仪、信号发生器等，为用户带来极大的方便，极大地提高了应用系统设计效率。

2.1.1　Proteus 支持的 ARM 模型

Proteus 支持 ARM7TDMI 内核模型仿真，该内核模型主要是作为建模原型（modelling primitive），用于创建各种 ARM7 芯片的模型，如 LCP2000 系列芯片，不推荐用户直接在电路设计中使用 ARM7TDMI 内核，因为它灵活而复杂的管道总线结构使得对存储器和其他外设的接口变得极其复杂。

ARM7TDMI 内核模型具有一系列的属性，如表 2-1 所列。但必须注意，对于一个具体的芯片模型来说，并不是所有的属性都是有意义的。只有对于该具体芯片模型有意义的属性才会显示在元器件编辑对话框中。

表 2-1　ARM7TDMI 模型属性表

属　　性	默 认 值	描　　述
PROGRAM	—	定义一个或多个程序文件。文件内容将被导入到模型的内部程序存储器中。文件格式可以是 Intel Hex 格式（.hex），或者是 ELF 格式（.elf）。必须包含文件扩展名
CLOCK	1MHz	定义 CPU 时钟频率。考虑到效率问题，将不会对电路中的时钟电路进行仿真，因此，CPU 的时钟速度将由这个值单独决定
MODE_UNPREDICTABLE	1	定义当模型产生不可预知的结果时，是否生成仿真日志和警告信息
MODE_SUPPRESS_BUS	1	定义是否压缩访问外部存储器（EXTROM 和 EXTRAM）的总线周期。设置 MODE_SUPPRESS_BUS=0，将对每个总线周期进行完整的仿真，当然这会降低仿真速度

（续）

属　　性	默 认 值	描　　述
EXTROM	0x00000000-0x0000FFFF	定义一个或多个 ROM 存储器的地址范围，如果设置了 MODE_SUPPRESS_BUS=1，将在模型内部仿真对这个地址范围的读周期。默认值为从 0x00000000 地址开始的 64KB 的 ROM 空间
EXTRAM	0x00010000-0x0001FFFF	定义一个或多个 RAM 存储器的地址范围，如果设置了 MODE_SUPPRESS_BUS=1，将在模型内部仿真对这个地址范围的读写周期。默认值为从 0x00010000 地址开始的 64KB 的 RAM 空间
ELF_DEBUG	0	是否打开 ELF 装载器（loader）的诊断功能
TRACE_XXX（仅用于 LPC2100 系列）	0	使能 LPC2100 系列 CPU 外设接口的调试和诊断，XXX 可以是 SYSCON、SPI、VIC、UART0、UART1、RTC、SPI、SPI0、SPI1、I^2C 或者 WATCHDOG。把这些属性的值设置为 1，就打开了各个外设接口的诊断/调试功能，它将把诊断/调试信息记录到仿真日志中

目前，Proteus 支持 NXP 公司基于 ARM7TDMI 内核的 LPC2000 系列 ARM 处理器芯片：

- LPC2101—8KB ROM，2KB RAM，10bit ADC。
- LPC2102—16KB ROM，4KB RAM，10bit ADC。
- LPC2103—32KB ROM，8KB RAM，10bit ADC。
- LPC2104—128KB ROM，16KB RAM。
- LPC2105—128KB ROM，32KB RAM。
- LPC2106—128KB ROM，64KB RAM。
- LPC2138—128KB ROM，16KB RAM，10bit ADC。
- LPC2124—256KB ROM，16KB RAM，10bit ADC。
- LPC2131—32KB ROM，8KB RAM，10bit ADC。
- LPC2132—64KB ROM，16KB RAM，10bit ADC，1 x DAC。
- LPC2134—128KB ROM，16KB RAM，2 x 10bit ADC，1 x DAC。
- LPC2136—256KB ROM，32KB RAM，2 x 10bit ADC，1 x DAC。
- LPC2138—512KB ROM，32KB RAM，2 x 10bit ADC，1 x DAC。

LPC2000 系列 ARM 模型都是使用 ARM7TDMI 内核原型并添加了基于 DLL 模型的外设而构成的。当前版本的 LPC2000 系列 ARM 模型不含以下功能：

- UART 的 FIFO 模式没有建模，将在下一个版本完善。
- LPC2138 和 LPC2124 外部中断多路选择器的与或逻辑没有模型。
- boot-loader 没有模型，Reset 后，程序总是从 0x00000000 地址开始执行。
- 在板编程（ISP）和在线编程（IAP）没有模型。
- 通过 MAM 进行存储访问的时序是近似的，使能 MAM 后，模型假设 100%的命中率，否则访问 FLASH 存储器将使用 MAMTIM 说明的 CCLK 周期数。
- 访问 VBP 外设的指令时序是近似的。
- PLL 模型是完整的，但是在 60MHz 的 CCLK 下仿真不可能达到实时。Proteus VSM 最快的处理速度大约为 10MIPS（3GHz 的 PC）。
- PWM 模块仿真模型不完整。

2.1.2　Proteus ELF/SWARF 装载器

ELF/DWARF 的符号调试格式已经成为实事上的标准，它允许各种调试工具装载不同编译器创建的可执行文件。任何能够生成包含 DWARF2 调试信息 ELF 文件的 C/C++编译器，都可以通过 Proteus ELF/DWARF 装载器支持源码级调试。例如以下编译器：

- GNU ARM V3.3.1。
- Keil MDK for ARM。
- IAR EWARM。

源码级调试信息都是从 ELF/DWARF 中提取的，因此必须对编译器进行相关的设置，以产生包含 DWARF2 调试信息的 ELF 文件。ELF 文件的扩展名是“.ELF”，要装载 ELF 文件到 Proteus 模型中，需要设置模型的 PROGRAM 属性，包括完整的文件名和扩展名。

例如：PROGRAM=MYFILE.ELF

必须注意，一个 ELF 文件可能并不包含 DWARF 调试信息（即是存放源代码和内存变量的部分）。如果发现 Proteus 装载了 ELF 文件，但不能显示源代码或变量窗口，请检查仿真日志，如果在仿真日志中发现如下信息：

No“debug_line”section，cannot associate addresses to source lines.

则意味着编译器没有生成（或者根本就不能生成）DWARF2 的调试信息。另外，ELF 文件包含了 C 源代码文件的绝对路径，如果用户对编译器的整个创建项目进行了移动，很可能也会出现问题。

2.1.3　Proteus LPC2000 的调试窗口

Proteus 为各类处理器提供了各种各样的调试窗口，这些窗口可以从 ISIS 的菜单 Debug 下的子菜单中找到，对于 LPC2000 系列的处理器，主要的调试窗口有以下几类：

1．Flash 存储器窗口

Flash 存储器窗口显示当前加载到 Flash 中的内容，如图 2-1 所示。需要注意的是，调试过程中，并不能修改 Flash 的内容，它只起到一个显示与观察的作用。有多种显示方式，可以通过右键菜单修改。

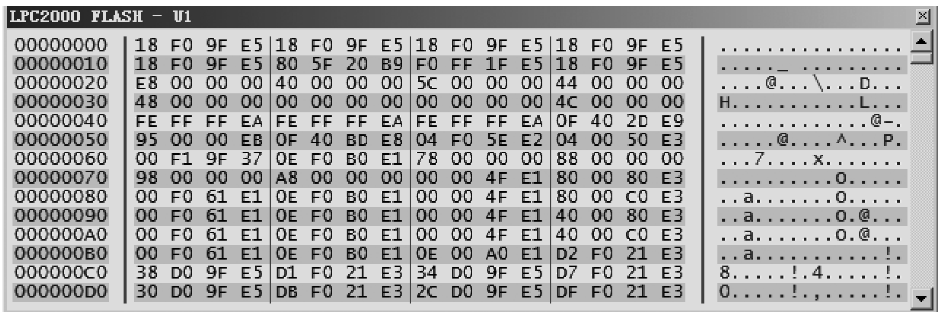


图 2-1　Flash 存储器窗口

2．RAM 存储器窗口

RAM 存储器窗口显示当前 RAM 存储器保存的内容，如图 2-2 所示。与 Flash 存储器窗

口一样，调试过程中也不能修改 RAM 中的内容，它只起到了一个显示与观察的作用。有多种显示方式，这可以通过右键菜单来修改。

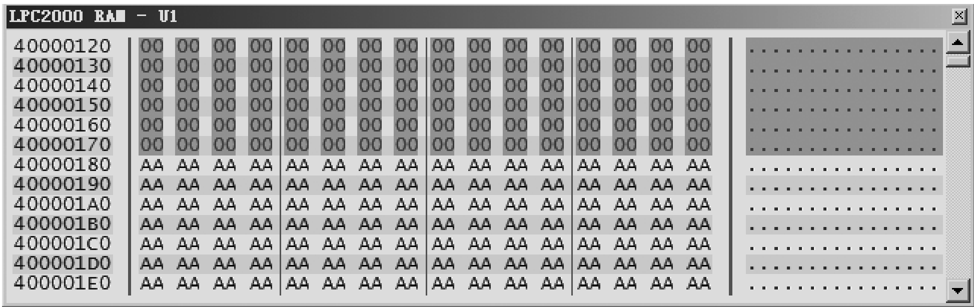


图 2-2　RAM 存储器窗口

3．观察窗口

观察窗口用于观察功能寄存器的值，但不能观察变量值，如图 2-3 所示，通过右键菜单，可以用寄存器名或者用寄存器的地址来添加用户需要的观察项。

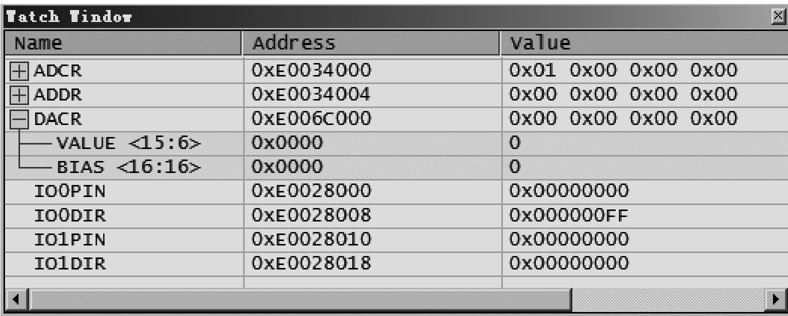


图 2-3　观察窗口

还可以设置条件断点，这给调试程序提供了很大的方便，如图 2-4 所示，但这仅限于功能寄存器，对于普通的存储器变量，是不能通过这种方法来添加观察点和设置条件断点的，而只能借助第三方集成开发环境，来设置变量观察点和条件断点。

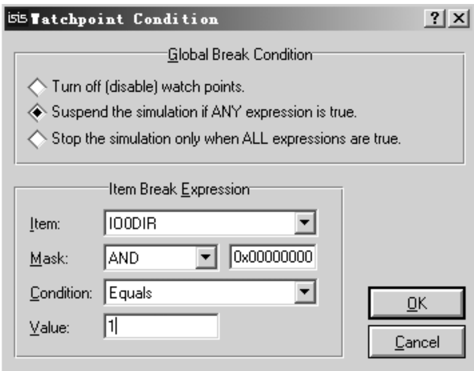


图 2-4　为功能寄存器设置条件断点

4．CPU 寄存器窗口

Proteus 还提供了一个 CPU 寄存器的观察窗口，如图 2-5 所示。其中显示了当前时钟周期数、当前执行的指令以及其他一些 CPU 寄存器内容，这些寄存器内容也是只读的，不能修改。

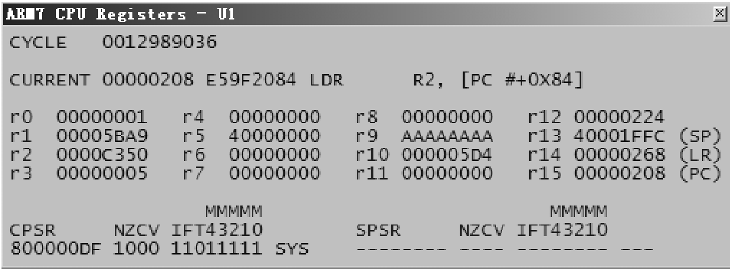


图 2-5　CPU 寄存器窗口

5．引脚配置窗口

引脚配置窗口显示当前的引脚配置，比较简单，如图 2-6 所示。

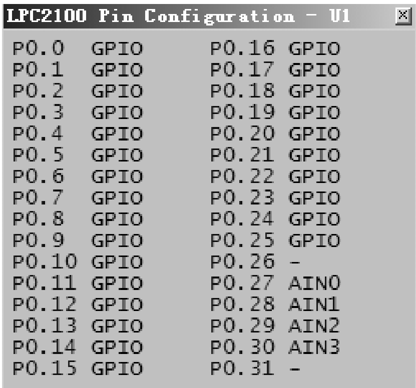


图 2-6　引脚配置窗口

6．仿真日志窗口

仿真日志窗口提供了丰富的实时仿真信息，如图 2-7 所示。结合 Proteus 提供的诊断器，可以方便地跟踪、调试 CPU 的各个功能模块的操作、状态等。

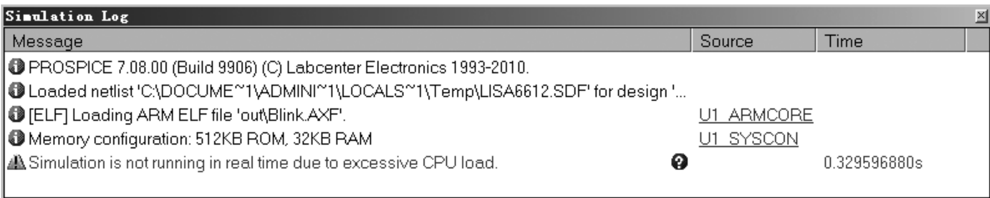


图 2-7　仿真日志窗口

诊断器是 Proteus 提供的一个新的功能，大大方便用户熟悉各个功能模块操作的细节。注意，诊断器不仅能跟踪 CPU、还可以跟踪其他器件，如 I^2C 操作等。这在电路调试和程序

调试中都非常有用。诊断器有 4 种跟踪等级：禁止跟踪（Disable）、限于警告（Warnings Only）、完全跟踪（Full Trace）和调试（Debug），如图 2-8 所示。

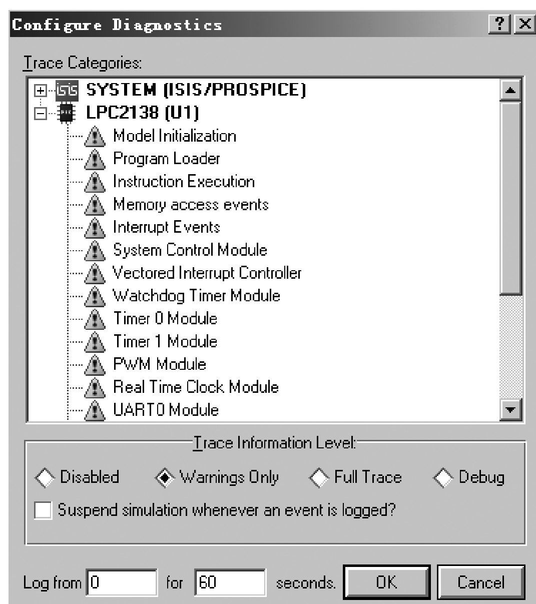


图 2-8　诊断器

对于 LPC2000 系列 ARM 处理器，可诊断主要部件有：

- Model Initialization —— 模型初始化。
- Porgram Loader —— 程序加载器。
- Instruction Execution —— 指令执行情况。
- Memory Access events —— 存储器访问事件。
- Interrupt Events —— 中断事件。
- System Control Module —— 系统控制模块。
- Vectored Interrupt Controller —— 向量中断控制器。
- Watchdog Timer Module —— 看门狗定时器模块。
- Timer0 Module —— 定时器 0 模块。
- Timer1 Module —— 定时器 1 模块。
- PWM Module —— PWM 模块。
- Real Time Clock Module —— 实时时钟模块。
- UART0 Module —— UART0 模块。
- UART1 Module —— UART1 模块。
- SPI0 Module —— SPI0 模块。
- SPI1 Module —— SPI1 模块。
- I^2C Module —— I^2C 模块。
- ADC Module —— ADC 模块。

2.2 集成环境 ISIS

Proteus 软件包提供一种界面友好的人机交互式集成环境 ISIS，其设计功能强大，使用方便。ISIS 在 Windows 环境下运行，启动后弹出如图 2-9 所示界面，由下拉菜单、快捷工具按钮、预览窗口、原理图编辑窗口、元器件列表窗口、元器件方向选择、仿真按钮组成。

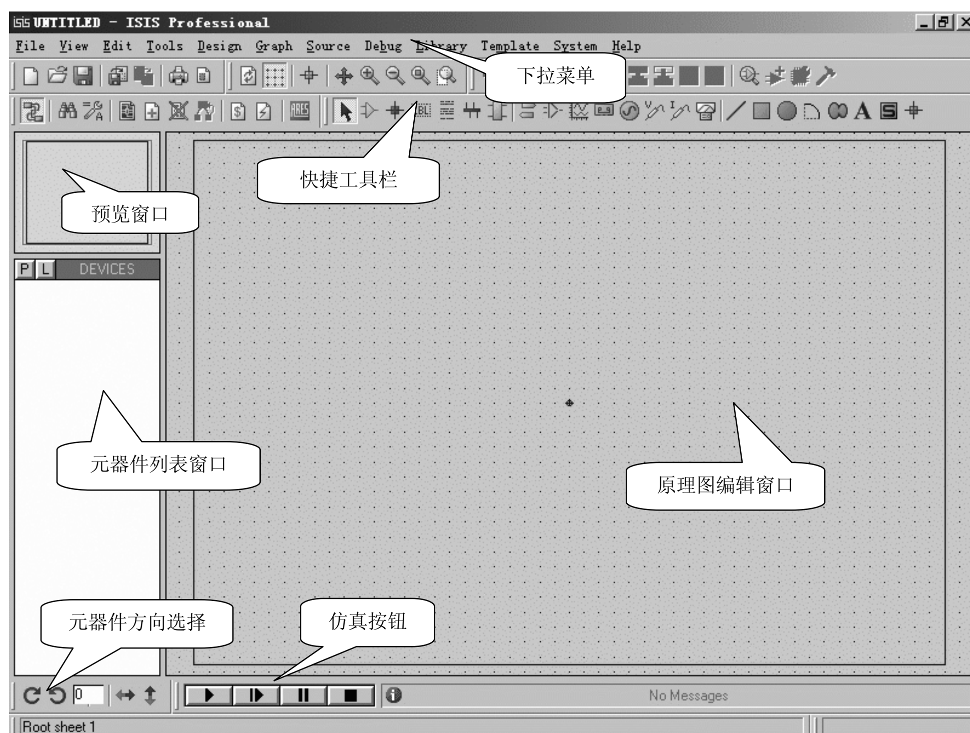


图 2-9 ISIS 环境界面

下拉菜单提供如下功能选项：

File 菜单包括常用的文件功能，如创建一个新设计、打开已有设计、保存设计、导入/导出文件、打印设计文档等。

View 菜单包括是否显示网格、设置网格间距、缩放原理图、显示与隐藏各种工具栏等。

Edit 菜单包括撤销/恢复操作、查找与编辑、剪切、复制、粘贴元器件、设置多个对象的层叠关系等。

Library 菜单包括添加、创建元器件/图标、调用库管理器。

Tools 菜单包括实时标注、实时捕捉、自动布线等。

Design 菜单包括编辑设计属性、编辑图纸属性、进行设计注释等。

Graph 菜单包括编辑图形、添加 Trace、仿真图形、一致性分析等。

Source 菜单包括添加/删除源程序文件、定义代码生成工具、调用外部文本编辑器等。

Debug 菜单包括启动调试、进行仿真、单步执行、诊断设置等。

Template 菜单包括设置图形格式、文本格式、设计颜色、节点形状等。

System 菜单包括设置环境变量、工作路径、图纸尺寸大小、字体、快捷键等。

Help 菜单包括版权信息，帮助文件、例程等。

快捷工具栏分为主工具栏和元器件工具栏。主工具栏包括文件工具、视图工具、编辑工具、设计工具 4 个部分，每个工具栏提供若干快捷按钮。

主工具按钮如图 2-10 所示，从左往右各按钮功能依次如下：

- 新建设计。
- 打开已有设计。
- 保存设计。
- 导入文件。
- 导出文件。
- 打印设计文档。
- 标识输出区域。

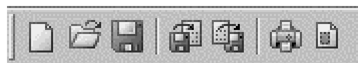


图 2-10　主工具按钮

视图工具按钮如图 2-11 所示，从左往右各按钮功能依次如下：

- 刷新。
- 网格开关。
- 原点。
- 选择显示中心。
- 放大。
- 缩小。
- 全图显示。
- 区域缩放。



图 2-11　视图工具按钮

编辑工具按钮如图 2-12 所示，从左往右各按钮功能依次如下：

- 撤销。
- 重做。
- 剪切。
- 复制。
- 粘贴。
- 复制选中对象。
- 移动选中对象。
- 旋转选中对象。
- 删除选中对象。
- 从器件库选元器件。
- 制作器件。
- 封装工具。
- 释放元件。



图 2-12　编辑工具按钮

设计工具按钮如图 2-13 所示，从左往右各按钮功能依次如下：

- 自动布线。
- 查找。
- 属性分配工具。

- 设计浏览器。
- 新建图纸。
- 删除图纸。
- 退到上层图纸。
- 生成元件列表。
- 生成电器规则检查报告。
- 创建网络表。



图 2-13　设计工具按钮

元器件工具栏包括方式选择、配件模型、绘制图形 3 个部分，每个工具栏提供若干快捷按钮。

方式选择按钮如图 2-14 所示，从左往右各按钮功能依次如下：

- 选择即时编辑元件。
- 选择放置元件。
- 放置节点。
- 放置网络标号。
- 放置文本。
- 绘制总线。
- 放置子电路图。



图 2-14　方式选择按钮

配件模型按钮如图 2-15 所示，从左往右各按钮功能依次如下：

- 端点方式，有 VCC、地、输出、输入等。
- 器件引脚方式，用于绘制各种引脚。
- 仿真图表。
- 录音机。
- 信号发生器。
- 电压探针。
- 电流探针。
- 虚拟仪表。

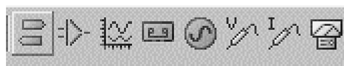


图 2-15　配件模型按钮

图形绘制按钮如图 2-16 所示，从左往右各按钮功能依次如下：

- 绘制直线。
- 绘制方框。
- 绘制圆。
- 绘制圆弧。
- 绘制多边形。
- 编辑文本。
- 绘制符号。
- 绘制原点。



图 2-16　图形模型按钮

在元器件列表窗口下方有一个元器件方向选择栏，其按钮如图 2-17 所示，从左往右各按钮功能依次如下：

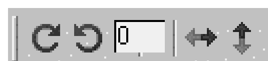


图 2-17　元器件方向选择按钮

- 向右旋转 90°。
- 向左旋转 90°。

● 水平翻转。

● 垂直翻转。

在原理图编辑窗口下方有一个仿真工具栏，其按钮如图 2-18 所示，从左往右各按钮功能依次如下：

● 全速运行。

● 单步运行。

● 暂停。

● 停止。

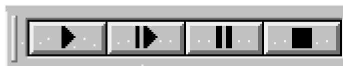


图 2-18　仿真工具按钮

原理图编辑窗口是用来绘制原理图的，蓝色方框内为编辑区，里面可以放置元器件和进行连线。注意，这个窗口没有滚动条，需要用预览窗口来改变原理图的可视范围，也可以用鼠标滚轮对显示内容进行缩放。

预览窗口可显示两种内容：一种是在元器件列表窗口选中某个元件时，将显示该元件的预览图；另一种是当鼠标落在原理图编辑窗口时（即放置元件到原理图编辑窗口后或在原理图编辑窗口中单击鼠标后），将显示整张原理图的缩略图，并会显示一个绿色的方框，绿色方框里面就是当前原理图编辑窗口中显示的内容，可用鼠标改变绿色的方框的位置，从而改变原理图的可视范围。

2.3　原理图绘制与源代码仿真调试

绘制原理图是在原理图编辑窗口中的蓝色方框内完成的，通过下拉菜单 System 中的 Set Sheet Size 选项，可以调整原理图设计页面大小。绘制原理图时，首先应根据需要选取元器件，Proteus ISIS 库中提供了大量元器件原理图符号，利用 Proteus ISIS 的搜索功能，可以很方便地查找需要的元器件。下面以图 2-19 为例来说明绘制原理图的方法。

首先根据需要选择器件。单击元器件列表窗口上边的按钮 P，弹出如图 2-20 所示元器件选择窗口，在该窗口左上方的“Keywords”栏内键入 LPC，窗口中间的“Results”栏将显示出元器件库中所有 LPC2000 ARM 处理器，选择其中的 LPC2138，窗口右上方的“LPC2138 Preview”栏将显示出图形符号，同时显示该器件的虚拟仿真模型（LPC213x），单击“OK”后，选择的器件将出现在器件列表窗口。照此办理，选择所有需要的元器件，如果选择的器件显示“No Simulator Model”，说明该器件没有仿真模型，将不能进行虚拟仿真。

如果遇到库中没有的器件，就需要自己创建。通常有两种方法创建自己的元件：一种是用 PROTEUS VSM SDK 开发仿真模型，并制作元件；另一种是在已有的元件基础上进行改造。关于具体创建方法这里不作介绍，请读者查阅相关资料。

器件选择完毕后，就可以开始绘制原理图了。先用鼠标从器件选择窗口选中需要的器件，预览窗口将出现该器件的图标。再将鼠标指向编辑窗口并单击左键，将选中的器件放置到原理图中。

放置电源和地线端时，要从配件模型按钮栏中选取。

在两个器件之间进行连线的方式很简单，先将鼠标指向第一个器件的连接点并单击左键，再将鼠标移到另一个器件的连接点并单击左键，这两个点就被连接到了一起。对于相隔较远，直接连线不方便的器件，可以用标号的方式进行连接，如图 2-19 中按键 KEY1～

KEY8 与 LPC2138 的 P0.8～P0.15 各口线之间就是通过标号相连的，需要注意的是，如果使用总线方式标明连接点，真正起作用的是标号，而总线只是一个标识符号而已。

图 2-19　绘制原理图示例

图 2-20　器件选择窗口

在编辑窗口中绘制原理图的一般操作总结如下：用左键放置元件，右键选择元件，双击右键删除元件，右键拖选多个元件，先右键后左键编辑元件属性，先右键后左键拖动元件，连线用左键，删除用右键，中键滚轮缩放整个原理图。

原理图绘制完成后，还要给 ARM 处理器装入应用程序，才可以进行虚拟仿真调试。先用鼠标右键选中 LPC2138，再单击左键，弹出如图 2-21 所示器件编辑窗口。

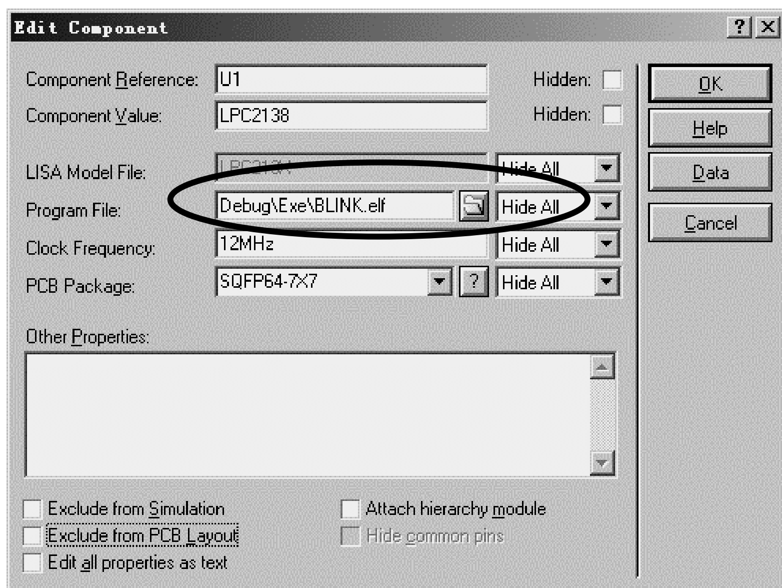


图 2-21　器件编辑窗口

在器件编辑窗口中“Program File”栏单击文件夹浏览按钮，找到需要仿真的 Hex 文件或 ELF 文件，单击“OK”按钮完成可执行文件装入，这时单击仿真工具栏中全速运行按钮即可开始进行虚拟仿真。需要注意的是，装入到原理电路图 LPC2138 中的 Hex 文件，可以进行虚拟仿真并观察程序运行结果，但不能进行源代码调试，只有装入的是包含 DWARF2 调试信息的 ELF 文件，才能进行源代码调试。

为了直观看到仿真过程，还可以在原理图中添加一些虚拟仪表，可用的虚拟仪表有：电压表、电流表、虚拟示波器、逻辑分析仪、计数器定时器、虚拟终端、虚拟信号发生器、序列发生器、I^2C 调试器、SPI 调试器等。

先点击仿真工具中的暂停按钮，再单击 Debug 下拉菜单中的“ARM7 Source Code”选项，弹出如图 2-22 所示源代码调试窗口。源代码调试窗口右上角提供的调试按钮功能如下：

全速运行（Run）。启动程序全速运行。

单步运行（Step Over）。执行子程序调用指令时，将整个子程序一次执行完。

跟踪运行（Step Into）。遇到子程序调用指令时，跟踪进入子程序内部运行。

跳出运行（Step Out）。将整个子程序运行完成，并返回到主程序。

运行到光标处（Run To）。从当前指令运行到光标所在位置。

设置断点（Toggle Breakpoint）。将光标所在位置设置一个断点。



图 2-22　源代码调试窗口

将鼠标指向源代码调试窗口并单击右键，弹出如图 2-23 所示右键快捷菜单，提供如下功能选项：

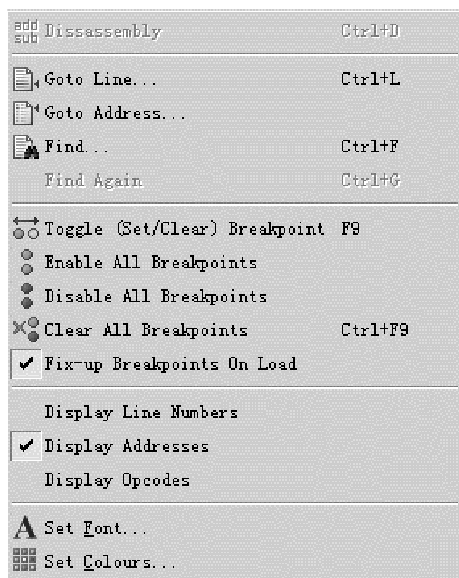


图 2-23　源代码调试窗口中的右键菜单

- “Goto Line…”，单击该选项，在弹出的对话框中键入源程序代码的行号，光标立即跳转到指定行。
- “Goto Address…”，单击该选项，在弹出的对话框中键入源程序代码的地址，光标立即跳转到指定地址处。
- “Find…”，单击该选项，在弹出的对话框中键入希望查找的文本字符，将在源代码调

试窗口从当前光标所在位置开始查找指定的字符。

- “Find Again…”，将重复上次查找内容。
- “Toggle Breakpoint”，在光标所在处设置或删除断点。
- “Enable All Breakpoints”，允许所有断点。
- “Disable All Breakpoints”，禁止所有断点。
- “Clear All Breakpoints”，清除所有断点。
- “Fix–up All Breakpoints On Load”，装入时修复断点。
- “Display Line Numbers”，显示行号。
- “Display Addresses”，显示地址。
- “Display Opcode”，显示操作码。
- “Set Font…”，单击该选项，在弹出的对话框中设置源代码调试窗口中显示字符的字体。
- “Set Colour…”，单击该选项，在弹出的对话框中设置弹出窗口的颜色。

在 Proteus 中进行源代码调试时，Debug 下拉菜单提供了多种弹出式调试窗口（已在前面 2.1.3 节中介绍），给调试过程带来了许多方便。

2.4 用 IAR EWARM 生成源代码仿真调试程序

瑞典 IAR 公司多年来致力于 C 语言编译器的研究，该公司开发的 IAR EWARM 是一种专为 ARM 处理器设计的高效率 C 语言编译环境，包括项目管理、程序编译、连接定位等，IAR EWARM C 编译器，符合 ANSI 标准，能够生成包含 DWARF2 调试信息的 ELF 文件，可以很方便地装入 Proteus 原理图进行源代码仿真调试，为 ARM 嵌入式系统开发带来极大的方便。下面通过一个实例来说明采用 IAR EWARM 环境生成 Proteus 源代码仿真调试程序的方法和步骤。

2.4.1 创建项目

在 IAR EWARM 集成开发环境中用户需要建立一个工作区（workspace），用于创建项目，每个项目都可以建立以组（group）为级别的结构，用户源程序文件可以分别添加到各个组中。通过不同选项可以对项目进行灵活配置。建议用户先在硬盘上建立一个特定目录，用来存放各种与项目有关的文件，下面的例子采用的目录是 D:\GPIO 流水灯。

进入 IAR EWARM 集成开发环境后，单击 File 下拉菜单中的“New”→“Workspace”选项，就创建了一个工作区，如图 2-24 所示。

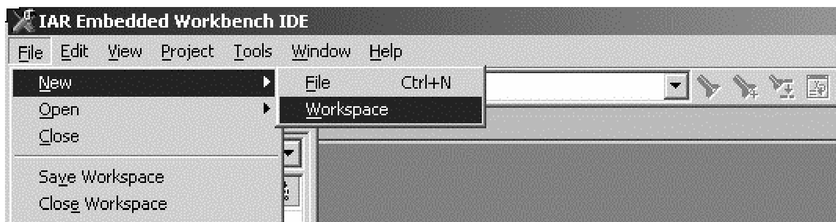


图 2-24　创建工作区

接下来要创建新项目。单击 Project 下拉菜单中的 Create New Project 选项，弹出创建新项目窗口，如图 2-25 所示。EWARM 提供了几种不同应用程序和库程序的项目模板，用户可以按照模板创建自己的新项目。

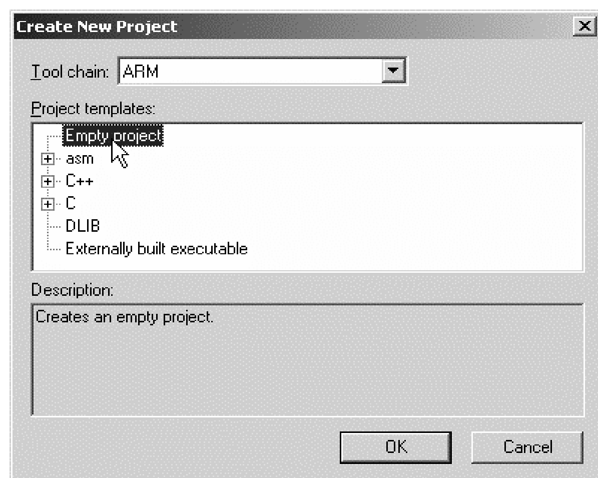


图 2-25　创建新项目窗口

在 Project Template（项目模板）栏内，选择 Empty project，表示采用默认的项目配置。在 Tool chain 栏内选择 ARM，表示当前使用 ARM 处理器。单击“OK”按钮后，弹出保存项目窗口，如图 2-26 所示。



图 2-26　保存项目窗口

在文件名栏内键入 BLINK，单击“保存”按钮，就创建并保存了一个新项目，该项目将出现在工作区窗口中，如图 2-27 所示。项目名称中的星号表示当前的修改还没有保存。默认状态下，系统将自动生成两个配置：调试（Debug）和发布（Release），用户可通过工作区窗口顶部的下拉菜单选择配置选项，本例中只使用 Debug 配置。至此就在 D:\GPIO 流水灯\目录下创建了一个项目文件，其文件扩展名为.ewp，其中包含了用户对项目的各种特殊设定。

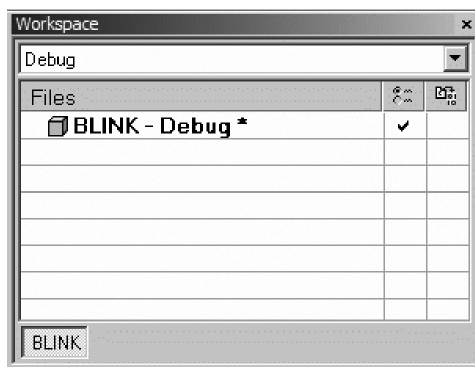


图 2-27　工作区窗口

最后还需要保存工作区，单击 File 下拉菜单中的 Save Workspace 选项，弹出保存工作区对话框，如图 2-28 所示。选定工作区文件的存放路径后（本例选择新建立的 D:\GPIO 流水灯\目录），在文件名栏内键入 BLINK.eww，单击“保存”按钮，就在选定的目录下创建了一个工作区文件 Fib.eww。这个文件中列出了所有加入到工作区的项目，其他相关信息都存放在 D:\GPIO 流水灯\settings 目录下。

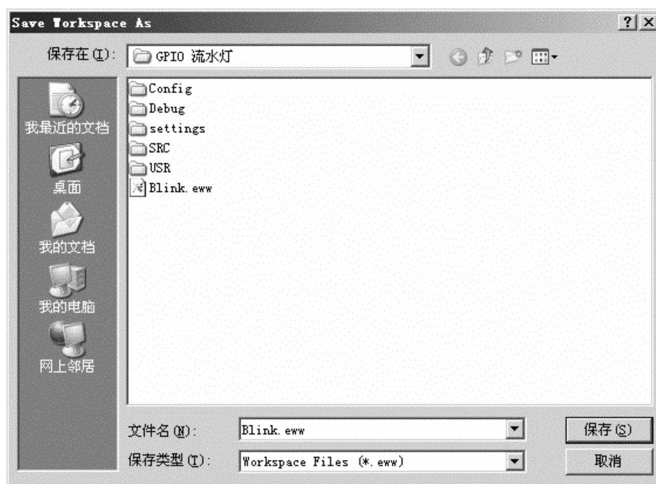


图 2-28　保存工作区窗口

保存工作区后，下一步就是在项目中新建文件或添加已有文件。项目中允许文件分组，用户可以根据项目需要和自己的习惯来组织源文件。本例新建 USR 和 SRC 两个文件组，USR 文件用来保存用户主程序，SRC 文件组用来保存其他配置文件等。单击 Project 下拉菜单中的 Add Group…选项，添加 USR 和 SRC 文件组，如图 2-29 所示。

新建源文件时单击 File 下拉菜单中的“New”→“File”选项，从打开的编辑窗口中输入源程序文件 main.c 和 Target.c，输入完毕后，分别保存到硬盘 D:\GPIO 流水灯\USR 和 D:\GPIO 流水灯\SRC 目录下。将鼠标指向工作区窗口的 USR 文件组并单击右键，从弹出的右键菜单中选择 Add Files…选项，将硬盘 D:\GPIO 流水灯\USR 目录下的 main.c 文件添加到 USR 文件组中，再用同样的方法将硬盘 D:\GPIO 流水灯\SRC 目录下的 Target.c 文件添加到

SRC 文件组中，如图 2-30 所示。

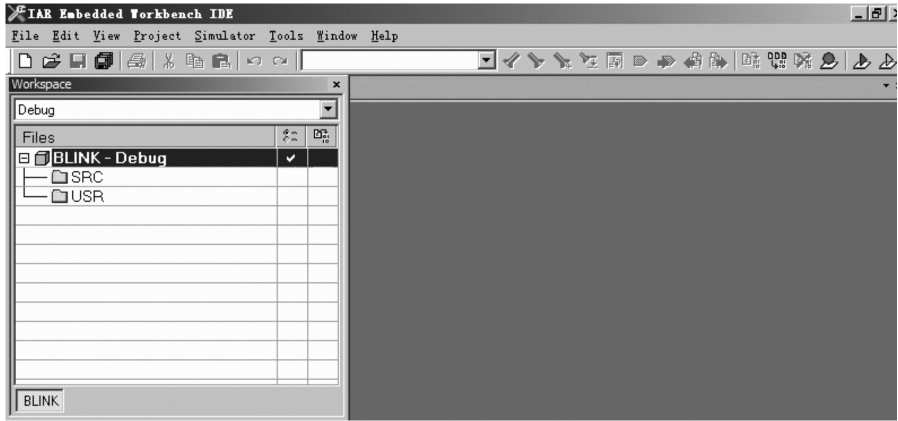


图 2-29　在工作区窗口新建两个文件组

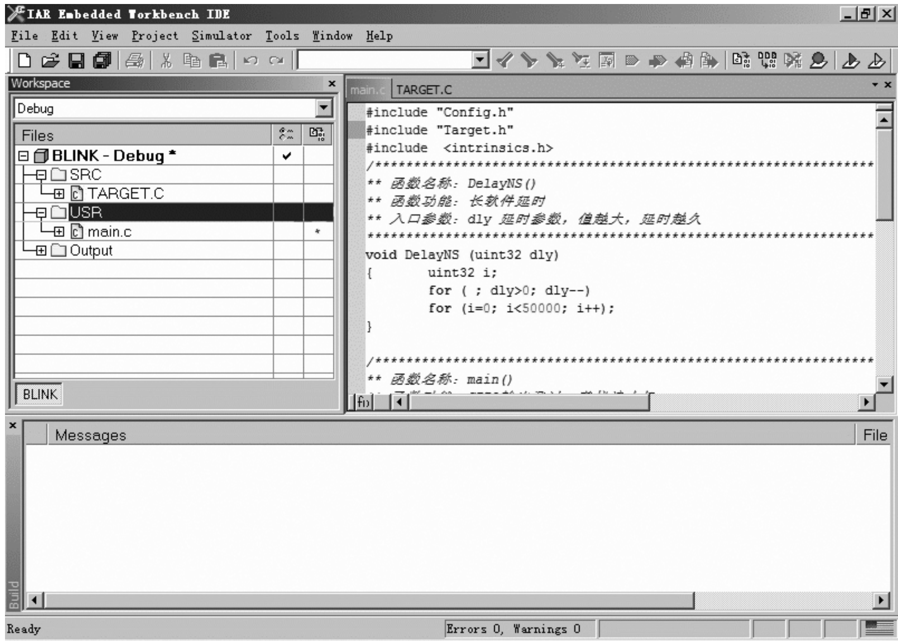


图 2-30　为工作区文件组添加文件

需要注意的是，往项目中添加文件时，只需添加汇编语言和 C 语言源程序，不需要添加头文件（即.h 头文件）。但是用户必须在配置项目的编译器、链接器选项时指明包含头文件的路径和目录。

2.4.2　项目选项配置与编译链接

创建新项目和添加文件后还需要为项目配置选项。配置项目选项是非常重要的一步，如果配置不当，编译、链接就会出错而无法生成正确的程序代码。前面在创建新项目时我们选择了 Empty project 模板，表示采用默认的项目选项配置，这些默认配置还要根据具体

项目进行修改。IAR EWARM 提供的项目选项内容繁多，初学者可能会感觉无从下手。实际上只要把关键的选项配置正确了，其他的采用默认配置就不会出错，下面介绍这些关键的选项配置。

先在工作区窗口选中项目文件夹 BLINK-Debug，单击 Project 下拉菜单中的 Options 选项，在弹出的 Options 窗口的左边 Category 栏中选择第一项 General Options，然后再选择右边的 Target 页，在 Processor Variant 栏中选择 Device，并单击右边的器件选择按钮，选择芯片型号 NXP LPC2138，同时在 Endian mode 栏中选择 Little，如图 2-31 所示。

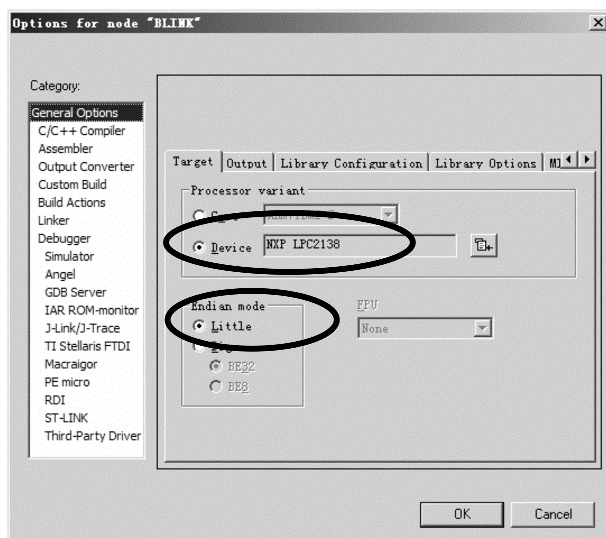


图 2-31　通用选项配置

在 Options 窗口的左边 Category 栏中选择第二项 C/C++ Compiler，然后再选择右边的 Code 页，在 Processor mode 栏中选择 ARM，如图 2-32 所示。

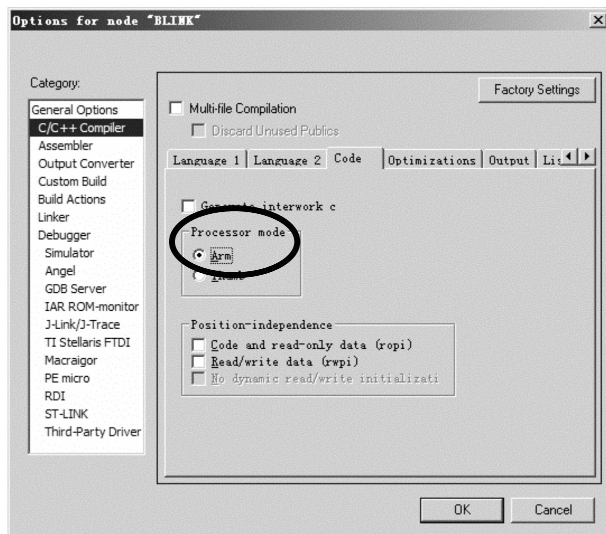


图 2-32　编译器 Code 选项配置

在 Options 窗口的左边 Category 栏中选择第二项 C/C++ Compiler，然后再选择右边的 Optimizations 页，在 Level 栏中选择 None，如图 2-33 所示。

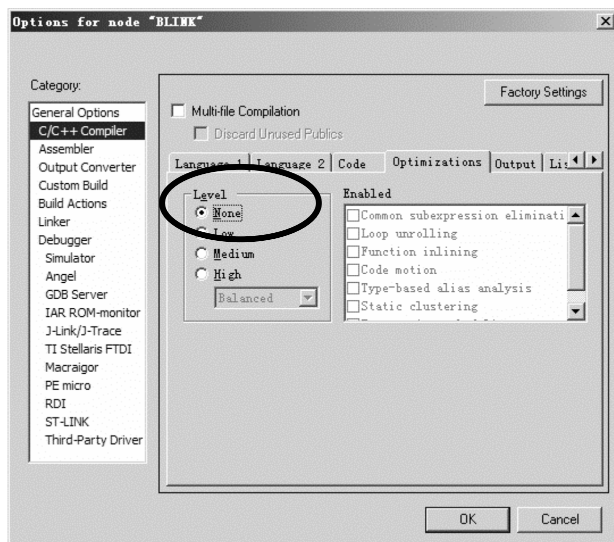


图 2-33 编译器 Optimizations 选项配置

在 Options 窗口的左边 Category 栏中选择第二项 C/C++ Compiler，然后再选择右边的 Preprocessor 页，该页面中列有标准的 include 文件目录。如果用户的 include 文件不在标准目录下，则必须在 Additional include directories 栏中键入包含该项目 include 文件的目录，一个目录用一行描述，本例键入的是$TOOLKIT_DIR$\INC\NXP 和$PROJ_DIR$\SRC，前者为 IAR 安装目录下软件自带的头文件，如 iolpc2138.h 头文件就在此目录下，后者代表当前项目的 SRC 目录，如项目配置头文件 CONFIG.h 和 Target.h 就在此目录下，如图 2-34 所示。

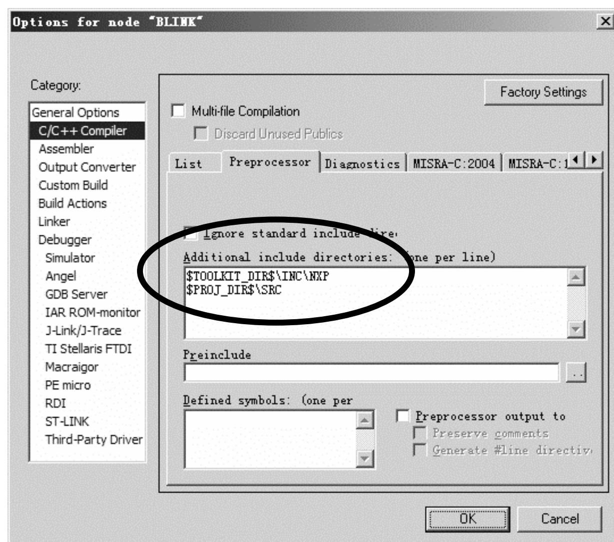


图 2-34 编译器 Preprocessor 选项配置

在 Options 窗口的左边 Category 栏中选择第四项 Output Converter，然后在右边的 Output 页中选择 Generate additional output，并在 Output format 对话框中选择 Intel extended，如图 2-35 所示。

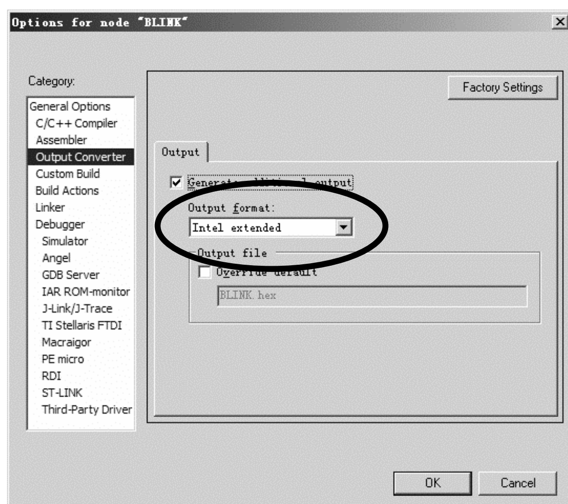


图 2-35　输出转换选项配置

在 Options 窗口的左边 Category 栏中选择第七项 Linker，然后再选择右边的 Config 页，选择 Linker configuration file 栏中的 Override default，并在下面的对话框中键入 $PROJ_DIR$\Config\Flash.icf，如图 2-36 所示。

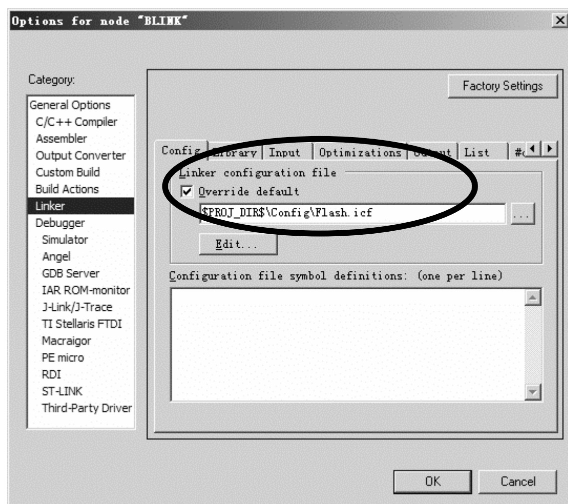


图 2-36　链接器 Config 选项配置

在 Options 窗口的左边 Category 栏中选择第七项 Linker，然后再选择右边的 Output 页，Output 栏中的默认文件名为 BLINK.out，如果希望生成带源代码仿真调试信息的输出文件，则应在 Output 栏中键入 BLINK.elf，并选择下面的 Include debug information in output 选项，如图 2-37 所示。

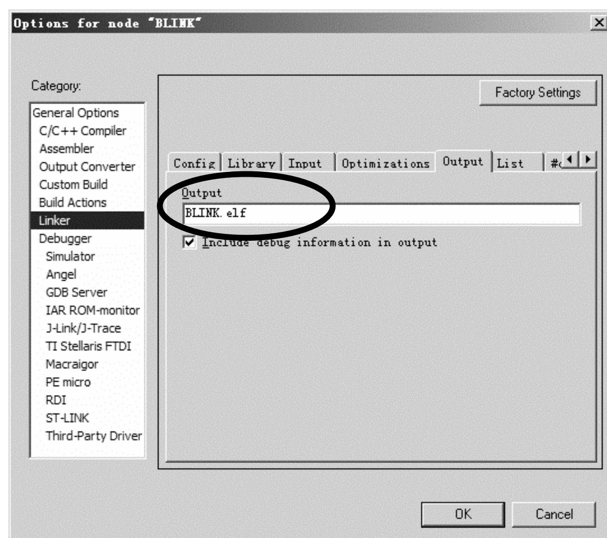


图 2-37　链接器 Output 选项配置

至此，关键项目配置已经完成，下面就可以进行项目编译了。单击 Project 下拉菜单中的 Make 选项，对项目中的全部源程序进行编译。编译过程产生的各种信息都将显示在屏幕下方的 Build 窗口中。如果源程序存在错误，会在 Build 窗口显示相应的提示信息，将鼠标指向窗口显示错误的地方并双击左键，光标将自动跳转到源程序文件产生错误的地方以便于修改，修改完成后，编译正确，即可生成带源代码调试信息的 ELF 文件。Project 下拉菜单中的 Make 选项只对项目中修改过的文件进行编译链接，这样可以加快执行速度，如果需要重新编译链接项目中所有的文件，则应选择 Project 下拉菜单中的 Rebuild All 选项来完成。

从编译链接后的工作区窗口中的树结构中，可以看到每个源文件访问关联了哪些头文件，同时生成了哪些输出文件。因为前面在建立新项目时选择了 Debug 配置，所以在 D:\GPIO 流水灯\目录下自动生成一个 Debug 子目录，Debug 子目录下又包含另 3 个子目录，分别为 List、Obj 和 Exe，程序编译链接成功后，将在 EXE 目录下生成 BLINK.out、BLINK.hex 和 BLINK.elf 文件，其中 hex 和 elf 文件都可于 Proteus 仿真，但只有 elf 文件才能进行源代码仿真调试。

复习思考题

1．Proteus 支持 NXP 公司的哪些 LPC2000 系列 ARM 处理器芯片？
2．Proteus 支持哪些编译器？
3．Proteus 为 ARM 处理器提供了哪些调试窗口？
4．Proteus 诊断器对 ARM 处理器提供了哪些可诊断部件？
5．在 Proteus 中绘制如图 2-19 所示的 LPC2138 应用系统原理电路。
6．如果希望进行源代码仿真调试，应给原理电路图 LPC2138 中装入什么文件？
7．对照图 2-19，采用 IAR EWARM 集成开发环境编写应用程序，并进行源代码仿真。

第 3 章　LPC213x ARM 处理器

3.1　LPC213x 处理器特性与存储器结构

3.1.1　主要特性

LPC213x 是 NXP 公司推出的一款基于 ARM7TDMI-S I 内核的 32 位 ARM 处理器，具体芯片包括 LPC2131/32/34/36/38 等，它们的功能特性大同小异，主要差别在于存储器空间大小以及片内集成外设功能配置。为简单起见，本章主要介绍 LPC2138 芯片，它带有 512KB 的片内高速 Flash 和 32KB 的片内 SRAM 存储器，内置了宽范围的串行通信接口（2 个 UART、SPI、SSP、I^2C）、多个 32 位定时器、2 个 10 位 ADC、1 个 10 位 DAC、PWM 通道、45 个高速 GPIO 以及多达 9 个边沿或电平触发的外部中断引脚。采用小型 LQFP64 封装和极低的功耗设计，使 LPC2138 特别适于工业控制、医疗系统、POS 机等小型应用，也非常适于通信网关、协议转换器、软件 modem、语音识别和低端成像系统等应用。

LPC2138的主要特性如下：

- 32 位 ARM7TDMI-S 微控制器核。32KB 的片内静态 RAM 和 512KB 的片内 Flash 存储器。128 位宽度的接口/加速器可实现高达 60MHz 工作频率。
- 通过片内 boot 装载程序，实现在系统编程（ISP）和在应用编程（IAP）。单个 Flash 扇区或整片擦除时间为 400ms。256B 编程时间为 1ms。
- EmbeddedICE RT 和嵌入式跟踪接口通过片内 RealMonitor 软件提供实时调试功能和高速跟踪指令执行功能。
- 2 个 10 位的 ADC 转换器，提供总共 16 路模拟输入，每个通道的转换时间为 2.44μs。
- 1 个 10 位的 DAC 转换器，产生可变的模拟量输出。
- 2 个 32 位定时器/外部事件计数器（带 4 路捕获和 4 路比较通道）、PWM 单元（6 路输出）和看门狗电路。
- 低功耗实时时钟（RTC）具有独立的电源和时钟源。
- 多个串行接口，包括 2 个 UART（16C550）、2 个高速 I^2C 总线（400Kbit/s）、SPI 和 SSP（具有缓冲功能，数据长度可变）。
- 向量中断控制器（VIC），可配置优先级和向量地址。
- 小型 LQFP64 封装，多达 45 个通用 I/O 口（可承受 5V 电压），多达 21 个边沿或电平触发的外部中断引脚。
- 片内集成振荡器，可使用外部晶体的频率范围为 1～25MHz。
- 通过一个可编程的片内锁相环 PLL（设置时间为 100μs）可实现最大为 60MHz 的

CPU 工作频率。

- 单电源供电，具有上电复位（POR）和掉电检测（BOD）电路。CPU 工作电压范围为 3.0～3.6V（3.3V±10%），I/O 端口可承受 5V 电压。
- 具有空闲和掉电低功耗模式。可通过单独允许/禁止外围功能和外围时钟分频，进一步实现功耗优化。可通过外部中断或 BOD 将处理器从掉电模式中唤醒。

图 3-1 所示为 LPC2138 的基本结构框图。

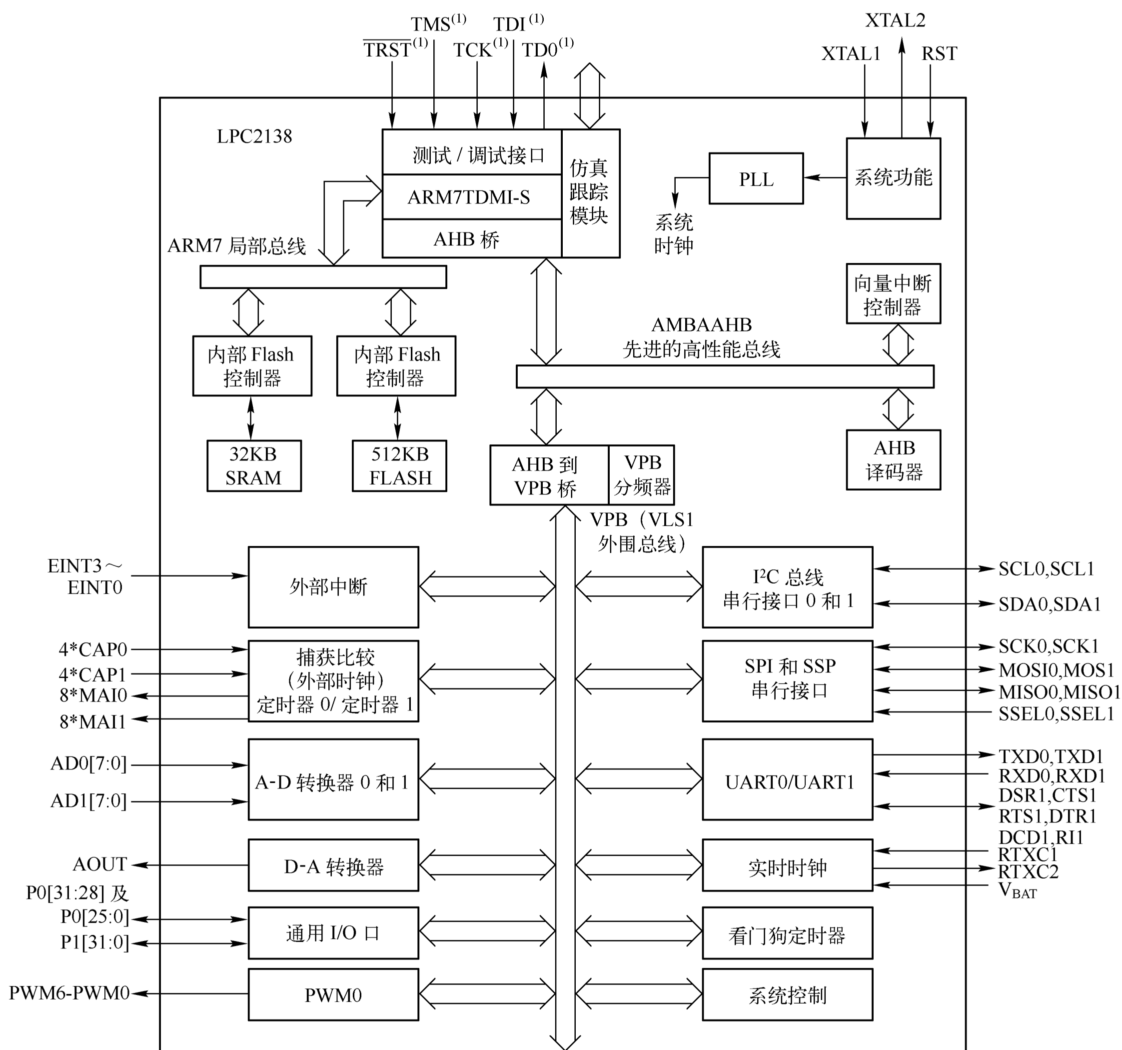


图 3-1 LPC2138 的基本结构框图

注：图中（1）表示引脚与 GPIO 共用。

3.1.2 存储器结构

LPC2138 集成了 512KB 的 Flash 存储器，可存储程序代码和数据。对 Flash 存储器的编程可通过串行口进行在系统编程（ISP），应用程序也可以在运行中擦除/编程 Flash，实现在

应用中编程（IAP），从而为数据保存和现场固件升级都带来极大的灵活性。Flash 存储器擦除/编程周期至少为 100 000 次，数据至少可保存 20 年。

LPC2138 具有 32KB 的静态 RAM，可存储程序代码或数据。SRAM 支持按 8 位、16 位和 32 位方式访问。LPC2138 复位后系统存储器地址空间映射如图 3-2 所示。此外，异常向量可以进行重新映射（Remap），允许它们位于 Flash 存储器（默认）或者片内静态 RAM 存储器中。



图 3-2　LPC2138 的系统存储器地址空间映射

LPC2138 的外设根据内部总线分为 AHB 和 VPB 外设两类，AHB 和 VPB 外设区域都为 2MB，可各自分配最多 128 个外设，每个外设空间的规格都为 16KB。所有外设寄存器不管规格大小，都按照字地址进行分配（32 位边界），并且不管字寄存器还是半字寄存器都是一次性访问。例如，不能对一个字寄存器的最高字节执行单独的读或写操作。

AHB 外设分配了 2MB 的地址范围（0xFFE0 0000～0xFFFF FFFF），它位于 4GB ARM 存储器空间的最顶端，VPB 外设也分配了 2MB 的地址范围（0xE000 0000～0xE01F FFFF），从 3.5GB 地址点开始。每个 AHB 外设和 VPB 外设都分配了 16KB 的地址空间。

3.1.3 存储器重新映射和 Boot Block

存储器映射的基本概念是：每个存储器组在存储器映射中都有一个物理上的位置，它是一个地址范围，在该范围内可以写入程序代码，每一个存储器空间的内容都永久固定在同一个位置，这样就不需要将代码设计成在不同地址范围内运行。ARM7 处理器的异常向量地址如表 3-1 所列。

表 3-1 ARM7 处理器异常向量地址

地　址	异　常	地　址	异　常
0x00000000	复位	0x00000010	数据中止
0x00000004	未定义指令	0x00000014	保留
0x00000008	软件中断	0x00000018	IRQ
0x0000000C	预取指中止	0x0000001C	FIQ

为了与将来器件相兼容，整个 Boot Block 都被映射到片内存储器空间的顶端。在这种方式下，使用较大或较小的 Flash 模块都不需要改变 Boot Block（需要改变 Boot 装载程序自身的代码）的位置或改变 Boot Block 异常向量的映射。除了异常向量之外的存储器空间都保持固定的位置。

存储器重新映射的部分允许在不同模式下处理中断，LPC2138 处理器支持 3 种存储器映射方式，如表 3-2 所列。当处理器工作在用户 Flash 模式下时，不需要进行异常向量的重新映射，而在其他模式下则需要重新映射。它包括异常向量区的 32B 和额外的 32B，一共 64B。重新映射的代码位置与地址 0x00000000～0x0000003F 相重叠。一个位于 Flash 存储器中的典型用户程序，可以将整个 FIQ 处理程序从地址 0x0000001C 开始存放，而不需要考虑存储器边界。在 SRAM、Flash 和 Boot Block 中的向量必须包含跳转到实际异常处理程序的分支或者其他执行跳转到异常处理程序的转移指令。

表 3-2 外设存储器地址空间映射

模　式	激　活	用　途
Boot 装载和程序模式	由任何复位硬件激活	在任何复位后都会执行 Boot 装载程序。Boot Block 异常向量映射到存储器的底部，以允许处理异常并在 Boot 装载过程中使用中断
用户 Flash 模式	由 Boot 代码软件激活	在存储器中识别了一个有效的用户程序标志且 Boot 装载操作未被执行时，由 Boot 装载程序激活。异常向量位于 Flash 存储器的底部，不需要重新映射
用户 RAM 模式	由用户程序软件激活	由用户程序激活，异常向量重新映射到 SRAM 的底部

选择这种配置有三个原因：

- 使 Flash 存储器中的 FIQ 处理程序不必考虑因为重新映射所导致的存储器边界问题。
- 用来处理代码空间中段边界仲裁的 SRAM 和 Boot Block 向量的使用大大减少。
- 为超过单字转移指令范围的跳转提供空间来保存常量。

重新映射的存储器组包括 Boot Block 和异常向量，除了重新映射的地址外，仍然继续出现在它们最初的位置。已重新映射和可重新映射区域的存储器地址空间如图 3-3 所示。

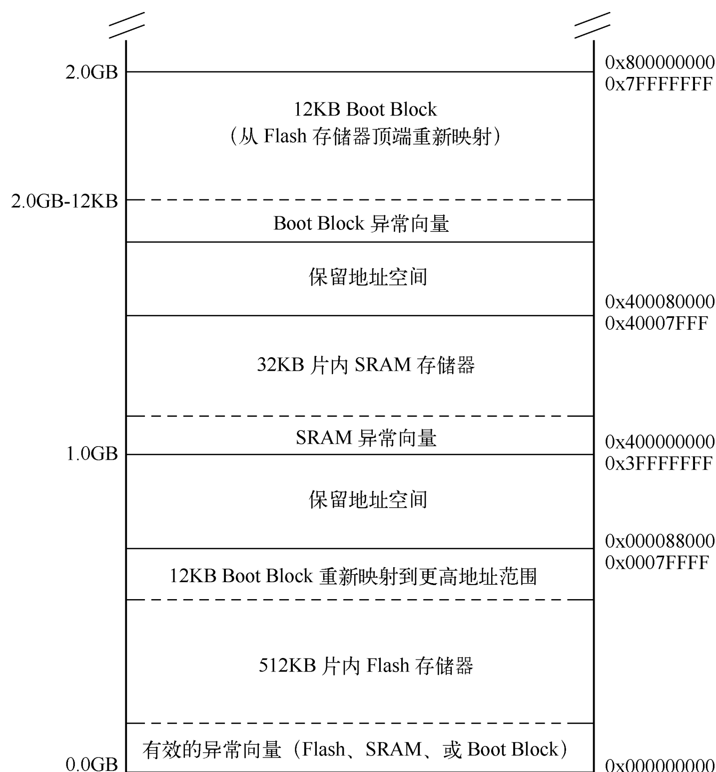


图 3-3　已重新映射和可重新映射区域的低存储器地址空间

存储器映射控制用于改变从地址 0x00000000 开始的异常向量映射，从而允许运行在不同存储器空间中的代码对异常进行控制，这是通过存储器映射控制寄存器 MEMMAP 来实现的。

MEMMAP 是一个可读/写寄存器（地址为 0xE01F C040），其功能为选择从 Flash Boot Block、用户 Flash 或 SRAM 中读取异常向量，如表 3-3 所列。

表 3-3　存储器地址映射控制寄存器 MEMMAP 功能

位	符　号	功 能 说 明	复 位 值
1:0	MAP	00：Boot 装载程序模式。异常向量从 BootBlock 重新映射 01：用户 Flash 模式。异常向量位于 Flash 中，不需要重新映射 10：用户 RAM 模式，异常向量从 SRAM 重新映射 11：保留，不使用	00
7:2	保留	未使用	NA

根据存储器映射的不同，异常向量的读取位置也不相同。存储器映射控制只从处理 ARM 异常必需的 3 个数据源（Flash 异常向量、SRAM 异常向量和 Boot Block 异常向量，每个 64B）中选择一个使用，如图 3-4 所示。

例如，每当产生一个软件中断请求时，ARM 内核就从 0x00000008 地址读取 32 位数据。这意味着当 MEMMAP[1:0]=00（Boot 装载程序模式）时，从 0x0000 0008 地址的读数/取指操作实际上是对 0x7FFFD008 地址单元进行操作。当 MEMMAP[1:0]=10（用户 RAM 模

式）时，从 0x00000008 地址的读数/取指操作实际上是对 0x40000008 地址单元进行操作。当 MEMMAP[1:0]=01（用户 Flash 模式）时，从 0x00000008 地址的读数/取指操作实际上是对片内 Flash 0x00000008 地址单元进行操作。

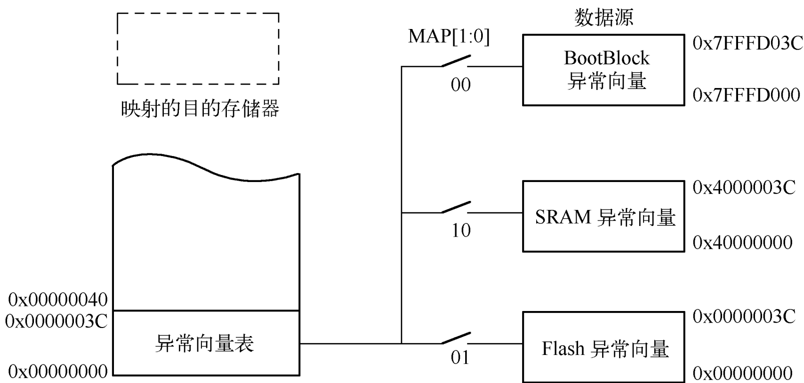


图 3-4　存储器映射控制

如果试图访问一个保留地址或未分配区域的地址，LPC21318 将产生适当的总线周期中止异常。这些区域包括：

- 特定的 ARM 器件所没有的存储器映射区域，在图 3-2 和图 3-3 中标志为“保留地址空间”。
- AHB 和 VPB 空间的保留区域。
- 未分配的 AHB 外设空间和未分配的 VPB 外设空间。

对于这些区域，对数据的访问和对指令的取指都会产生异常。此外，对 AHB 或 VPB 外设地址执行任何指令取指，都会导致产生预取指中止异常。

在现有的 VPB 外设地址空间内，对未定义地址的访问不会产生数据中止异常。每个外设内的地址译码被限制为外设内部需要判别的已定义寄存器。例如，对地址 0xE000D000（UART0 空间内一个未定义的地址）的访问可能导致对定义在地址 0xE000C000 处的寄存器进行访问。

需要注意的是，只有在试图执行从非法地址取指的指令时，ARM 才会将预取指中止标志与相关的指令（没有意义的指令）一起保存到流水线，并对中止进行处理（即读取指令时仅设置标志，指令实际执行时才产生中止）。当代码在非常靠近存储器边界执行时，这样可防止由预取指导致的意外中止。

3.1.4　存储器加速模块

LPC2138 处理器只使用一组 Flash 存储器，包含 3 个 128 位的缓冲区：

- 预取指缓冲区。
- 分支跟踪缓冲区。
- 数据缓冲区。

LPC2138 在运行位于 Flash 中的程序代码时，可以依靠存储器加速模块 MAM 最大限度的提高处理器性能。存储器加速模块 MAM 分为以下几个功能部件：

- 1 个 Flash 地址锁存器和 1 个增量器，用于预取指地址。
- 1 个 128 位的预取指缓冲区及其相关的地址锁存及比较器。
- 1 个 128 位的分支跟踪缓冲区及其相关的地址锁存及比较器。
- 1 个 128 位的数据缓冲区及其相关的地址锁存及比较器。
- 控制逻辑。
- 等待逻辑。

MAM 定义了 3 种操作模式：

① MAM 关闭模式。所有存储器请求都将导致 Flash 的读操作，没有指令预取。

② MAM 部分激活模式。如果数据可用，则从保持锁存区执行连续的指令访问，指令预取激活。非连续的指令访问启动 Flash 读操作，这意味着所有的转移指令都会导致从存储器取指。由于缓冲的数据访问时序很难预测，并且依赖于所处的状况，因此，所有数据操作都会导致 Flash 读操作。

③ MAM 完全激活模式。任何存储器请求（代码或数据），如果其值已经包含在其中一个保持锁存区中，那么从缓冲区执行该代码或数据访问，将导致指令预取激活。Flash 读操作用于指令的预取和当前缓冲区所没有的代码或数据访问。

处理器复位后，MAM 默认为禁止状态，软件可以随时将存储器访问加速功能打开或关闭。这样可以使大多数应用程序以最高速度运行，而某些要求更精确的功能，可以用较慢但可以预测的速度运行。启用 MAM 之后，Flash 编程功能不受 MAM 控制，而是作为一个独立的功能处理。

存储器加速模块 MAM 通过 2 个寄存器进行操作。

1．MAM 控制寄存器 MAMCR

MAMCR 寄存器用于决定 MAM 的操作模式。复位后 MAM 功能被禁止，改变 MAM 操作模式，将导致 MAM 所有的保持锁存内容无效，因此需要执行新的 Flash 操作。MAMCR 寄存器的功能如表 3-4 所列。

表 3-4　MAMCR 寄存器功能

位	功 能 说 明	复 位 值
1:0	00：MAM 功能被禁止 01：MAM 功能部分激活 10：MAM 功能完全激活 11：保留	00
7:2	保留	NA

2．MAM 定时寄存器 MAMTIM

MAMTIM 寄存器用于决定 Flash 存储器所使用的时钟个数（1～7 个处理器内核时钟 CCLK），从而可以调整 MAM 时序以便与处理器操作时序相匹配。MAMTIM 寄存器的功能如表 3-5 所列。

MAM 使用中应当注意以下两个问题：

（1）MAM 定时值问题

改变 MAM 定时值的时候，必须先通过向 MAMCR 寄存器写入 0 来关闭 MAM，然后再将新的值写入 MAMTIM 寄存器，最后将需要的操作模式对应值写入 MAMCR 寄存器，再次

打开 MAM。对于低于 20MHz 的系统时钟，建议将 MAMTIM 设定为 1；对于 20～40MHz 的系统时钟，建议将 MAMTIM 设定为 2；对高于 40MHz 的系统时钟，建议将 MAMTIM 设定为 3。

表 3-5　MAMTIM 寄存器功能

位	功 能 说 明	复 位 值
2:0	000：保留 001：MAM 取指周期为 1 个处理器内核时钟（CCLK） 010：MAM 取指周期为 2 个处理器内核时钟（CCLK） 011：MAM 取指周期为 3 个处理器内核时钟（CCLK） 100：MAM 取指周期为 4 个处理器内核时钟（CCLK） 101：MAM 取指周期为 5 个处理器内核时钟（CCLK） 110：MAM 取指周期为 6 个处理器内核时钟（CCLK） 111：MAM 取指周期为 7 个处理器内核时钟（CCLK）	0x07
7:3	保留	NA

（2）Flash 编程问题

在编程和擦除操作过程中不允许访问 Flash 存储器。如果在 Flash 模块忙时请求访问 Flash 地址，MAM 就必须强制 CPU 等待而导致代码执行的延时，这在某些情况下会导致看门狗超时而使 CPU 发生复位。用户必须注意这种可能性，采取一定的措施来保证不会在编程或擦除 Flash 存储器期间出现非预期的看门狗复位。

3.2　系统控制模块

3.2.1　时钟频率控制

LPC2138 处理器内部振荡电路支持频率为 1～50MHz 的时钟信号，如果使用了片内锁相环 PLL 或引导装载程序，输入时钟频率将被限制到 10～25MHz。LPC2138 对时钟频率 Fosc 的选择如图 3-5 所示。

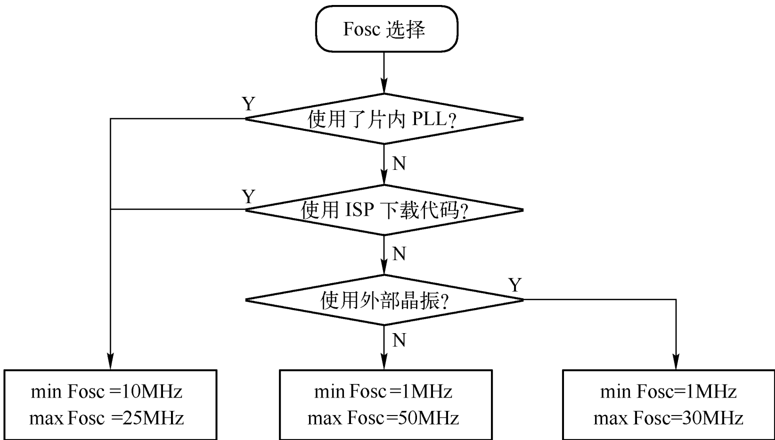


图 3-5　LPC2138 对时钟频率 Fosc 的选择

LPC2138 的振荡器可以工作于两种模式：从属模式和振荡模式。

从属模式如图 3-6a 所示，时钟信号与一个 100pF 的电容 Cc 相连后从 XTAL1 端输入，其幅值不低于 200mV，XTAL2 端不连接。从属模式下，要求时钟信号的占空比为 50%，频率限制在 1～50MHz。

振荡模式如图 3-6b 所示，由于片内集成了反馈电阻，只需在 XTAL1 和 XTAL2 端之间连接一个外部晶振 XTAL 和电容 Cx1、Cx2 就可形成基本模式的振荡电路。振荡模式下，晶振频率限制在 1～30MHz。

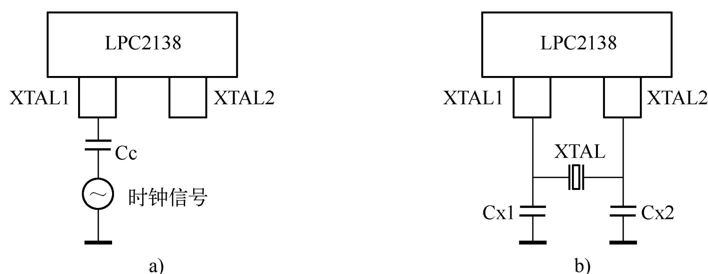


图 3-6　振荡器工作模式

a) 从属模式　b) 振荡模式

上述两种振荡器模式都可以用来驱动片内锁相环 PLL，给 CPU 和片内外设提供时钟。锁相环 PLL 接受输入时钟频率范围为 10kHz～25MHz，输入频率通过一个电流控制振荡器（CCO）倍增到范围 10～60MHz，倍频系数可以是从 1～32 的整数（实际上，由于 LPC2138CPU 最高频率限制为 60MHz，LPC2138 的倍频系数不能高于 6）。CCO 的操作频率范围为 156～320MHz，因此，锁相环中有一个额外的分频器，使得 PLL 在提供所需要的输出频率时，CCO 依然保持在允许频率范围内。输出分频器可设置为 2、4、8 或 16 分频，最小分频值为 2，保证 PLL 输出有 50%的占空比。

PLL 的激活由 PLLCON 寄存器控制，PLL 倍频器和分频器的值由 PLLCFG 寄存器控制。为了防止 PLL 参数发生意外改变或 PLL 失效，对这两个寄存器进行了保护。由于芯片的所有操作（包括看门狗定时器）都依赖于 PLL 提供的时钟，因此，PLL 设置的意外改变将导致 CPU 执行不期望的动作。对它们的保护是由一个类似于操作看门狗定时器的代码序列来实现的。

LPC2138 处理器通过如下几个寄存器来实现锁相环 PLL 控制。

1．PLL 控制寄存器 PLLCON

PLLCON 寄存器用于 PLL 的激活和连接。激活 PLL，将使 PLL 锁定到由当前倍频器和分频器的值设定的频率上。连接 PLL，将使处理器和所有片内功能都根据 PLL 的输出时钟频率运行。写入 PLLCON 寄存器的值只有在对 PLL 馈送寄存器 PLLFEED 执行了正确的 PLL 馈送序列之后才会生效。PLLCON 寄存器的功能如表 3-6 所列。

表 3-6　PLLCON 寄存器功能

位	功　能	功 能 说 明	复 位 值
0	PLLE	当 PLLE 为 1，并且在有效的 PLL 馈送之后，激活 PLL 并锁定到指定频率	0

（续）

位	功　能	功 能 说 明	复 位 值
1	PLLC	当 PLLE 和 PLLC 都为 1，并且在有效的 PLL 馈送之后，将 PLL 作为时钟源连接到处理器，否则处理器直接使用振荡器时钟	0
7:2	保留	未使用	NA

PLL 在作为时钟源之前必须进行设置、激活并锁定，PLL 在芯片复位和进入掉电模式时被关闭并旁路，PLL 只能通过软件激活，程序必须在配置并激活 PLL 后等待其锁定，然后再连接 PLL。将振荡器时钟切换到 PLL 输出或反过来操作时，内部电路对操作进行同步以确保不会产生干扰。硬件不能确保 PLL 在连接之前锁定或在 PLL 失去锁定时自动断开连接。在 PLL 失去锁定的情况下，振荡器很可能已经变得不稳定，这样断开 PLL 也挽救不了这种状况。

2．PLL 配置寄存器 PLLCFG

PLLCFG 寄存器用于控制倍频器和预分频器的值（分别用字母 M 和 P 表示），在执行正确的 PLL 馈送序列之前，写入 PLLCFG 寄存器的值不会生效。PLLCFG 寄存器的功能如表 3-7 所列。

表 3-7　PLL 配置寄存器功能

位	功　能	功 能 说 明	复 位 值
4:0	MSEL	PLL 倍频器的值，在 PLL 频率计算中其值为 M	0
6:5	PSEL	PLL 分频器的值，在 PLL 频率计算中其值为 P	0
7	保留	未使用	NA

3．PLL 状态寄存器 PLLSTAT

PLLSTAT 寄存器为只读寄存器，用于读回当前正在使用的 PLL 参数和状态。PLLSTAT 寄存器的值可能与 PLLCON 和 PLLCFG 的值不同，这是因为在没有执行正确的 PLL 馈送序列之前，PLLCON 和 PLLCFG 寄存器中的值无效。PLLSTAT 寄存器的功能如表 3-8 所列。

表 3-8　PLL 状态寄存器功能

位	功　能	功 能 说 明	复 位 值
4:0	MSEL	读出的 PLL 倍频器值，这是 PLL 当前使用的值	0
6:5	PSEL	读出的 PLL 分频器值，这是 PLL 当前使用的值	0
7	保留	未使用	NA
8	PLLE	读出的 PLL 激活位，为 1 表示 PLL 处于激活状态，为 0 表示 PLL 关闭，进入掉电模式时该位自动清零	0
9	PLLC	读出的 PLL 连接位，当 PLLE 和 PLLC 都为 1 时，将 PLL 作为时钟源连接到处理器。当 PLLE 和 PLLC 都为 0 时，PLL 被旁路，处理器直接使用振荡器时钟。进入掉电模式时该位自动清零	0
10	PLOCK	反映 PLL 锁定状态，为 0 时 PLL 未锁定，为 1 时锁定到指定频率	0
16:11	保留	未使用	NA

PLLSTAT 寄存器中的 PLOCK 位连接到中断控制器，这样可以使用软件打开 PLL 并连接到其他功能，不需要等待 PLL 锁定。发生中断时（PLOCK=1）可以连接 PLL 并禁

止中断。

PLL 有 3 种工作模式，由 PLLC 和 PLLE 位的组合控制，如表 3-9 所列。

表 3-9　PLL 的工作模式

PLLC	PLLE	PLL 功能
0	0	PLL 被关闭并断开连接，系统使用未更改的时钟输入
0	1	PLL 被激活但尚未连接，PLL 可在 PLOCK 置位后连接
1	0	与 00 组合相同，这样消除了 PLL 已连接但未使用的可能
1	1	PLL 被激活并连接到处理器作为系统时钟源

4．PLL 馈送寄存器 PLLFEED

必须将正确的馈送序列写入 PLL 馈送寄存器 PLLFEED，才能使 PLLCON 和 PLLCFG 寄存器的更改生效。馈送序列是将数值 0xAA 和 0x55 依次写入 PLLFEED 寄存器，写入顺序必须正确，并且必须在写入期间禁止中断。PLLFEED 寄存器的功能如表 3-10 所列。

表 3-10　PLLFEED 寄存器功能

位	功　能	功 能 说 明	复 位 值
7:0	PLLFEED	PLL 馈送序列必须写入该寄存器，才能使 PLL 配置和控制寄存器的更改生效	NA

如果用户程序要更改 PLL 设置，并且应用程序中开启了其他中断，则在重设 PLL 时（其实应该是发送 PLL 馈送序列时），需按照如下步骤进行：

- 关中断。
- 发送 PLL 馈送序列。
- 开中断。

LPC2138 处理器在掉电模式下会自动关闭并断开 PLL，从掉电模式唤醒不会自动恢复 PLL 的设定，必须由软件予以恢复。不要试图在掉电唤醒之后简单地执行馈送序列来重新启动 PLL，这将导致出现在 PLL 锁定之前同时激活并连接 PLL 的危险。

PLL 输出频率的计算会用到如下参数：

Fosc　　晶振频率

Fcco　　PLL 电流控制振荡器的频率

Fcclk　　PLL 输出频率（也是处理器的时钟频率）

M　　PLLCFG 寄存器中 MSEL 位的倍频值

P　　PLLCFG 寄存器中 PSEL 位的分频值

PLL 输出频率（当 PLL 激活并连接时）计算公式如下：

$$Fcclk = M \times Fosc \quad 或 \quad Fcclk = Fcco/(2 \times P)$$

CCO 频率计算公式如下：

$$Fcco = Fcclk \times 2 \times P \quad 或 \quad Fcco = Fosc \times M \times 2 \times P$$

PLL 的输入和设定必须满足下面条件：

- Fosc 的范围为 10～25MHz。
- Fcclk 的范围为 10MHz～Fmax（LPC2138 的最大允许频率）。

● Fcco 的范围为 156～320MHz。

PLL 的配置过程如下：

1）选择处理器工作频率 Fcclk。这可以根据处理器的整体要求、UART 波特率等因素来决定。外围器件的工作频率可以低于处理器频率器。

2）选择振荡器频率 Fosc。注意，Fcclk 必须是 Fosc 的整数倍。

3）计算 M 值以配置 MSEL 位。M = Fcclk/Fosc，M 取值范围为 1～32。写入 MSEL 位的值为 M-1，如表 3-11 所列。

4）选择 P 值以配置 PSEL 位，使 Fcco 在定义的频率限制范围内，Fcco 可通过前面介绍的公式计算。P 值必须是 1、2、4、或 8 中的一个。写入 PSEL 位的值所对应的 P 值如表 3-12 所列。

表 3-11　PLL 的倍频值

MSEL 位，PLLCFG[4:0]	M 值
00000	1
00001	2
00010	3
00011	4
...	...
11110	31
11111	32

表 3-12　PLL 的分频值

PSEL 位，PLLCFG[6:5]	P 值
00	1
01	2
10	4
11	8

下面举一个 PLL 配置实例。

假如系统要求 Fosc=10MHz，Fcclk=60MHz，PLL 配置如下：

M=Fcclk/Fosc=60MHz/10MHz=6，因此将 M-1=5 写入 PLLCFG[4:0]。

P=Fcco/(Fcclk×2)，Fcco 必须在 156～320MHz 之内。Fcco 取最低频率 156MHz，则 P=156MHz/(2×60MHz)=1.3，Fcco 取最高频率可得出 P=2.67。为了同时满足 Fcco 最低和最高频率要求，P 值只能为 2。因此，应写入 PLLCFG[6:5]=01。最终得到 PLLCFG 的值为 0x25。

3.2.2　VPB 分频器

VPB 分频器决定处理器时钟 Fcclk 与片内集成外设功能部件所使用时钟 Fpclk 之间的关系。VPB 分频器有两个用途：第一个是通过 VPB 总线为外设提供所需的 Fpclk 时钟，以便外设在合适的速度下工作。为了实现此目的，VPB 总线时钟频率可以降低到处理器时钟频率 Fcclk 的 1/2 或者 1/4。由于 VPB 总线必须在上电之后才能正常工作，因此，芯片复位后，VPB 总线时钟频率默认为处理器时钟频率 Fcclk 的 1/4。VPB 分频器的第二个用途是在应用不需要任何外设全速运行时使功耗降低。VPB 分频器与振荡器和处理器时钟的连接如图 3-7 所示。

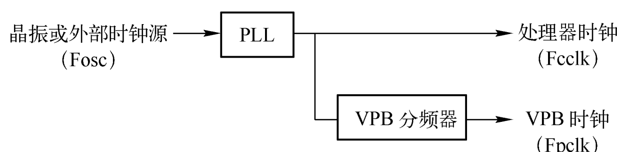


图 3-7　VPB 分频器的连接

VPB 分频器寄存器 VPBDIV

VPB 分频器寄存器 VPBDIV 用于控制 VPB 总线时钟频率，其功能如表 3-13 所列。

表 3-13　VPBDIV 寄存器功能

位	功　能	功 能 说 明	复 位 值
1:0	VPBDIV	VPB 总线时钟频率如下： 00：VPB 总线时钟为处理器时钟的 1/4 01：VPB 总线时钟与处理器时钟相同 10：VPB 总线时钟为处理器时钟的 1/2 11：保留。将该值写入 VPBDIV 寄存器无效（保留原来的设定）	00
7:2	保留	未使用	NA

3.2.3　功率控制

LPC2138 支持两种节电模式：空闲模式和掉电模式。

在空闲模式下，指令的执行被挂起直到发生复位或中断为止。外设功能在空闲模式下继续保持并可产生中断使处理器恢复运行。空闲模式下处理器、存储器系统和相关控制器以及内部总线不再消耗功率。任何中断都可以将 CPU 从空闲模式下唤醒。

在掉电模式下，振荡器关闭，芯片没有任何内部时钟。CPU 状态和工作寄存器、外设寄存器以及内部 SRAM 的内容在掉电模式下被保持。芯片引脚的逻辑电平保持静态。由于掉电模式使芯片所有的动态操作都被挂起，因此芯片的功耗降低到几乎为零。只有复位、外部中断、RTC 中断（RTC 必须使用外部独立的时钟源）和 BOD 中断可以将 CPU 从掉电模式下唤醒，使芯片恢复正常运行。

进入掉电和空闲模式必须与程序的执行相同步。通过中断唤醒掉电模式不会使指令丢失、不完整或重复。外设的功率控制特性允许独立关闭应用中不需要的外设，这样可以进一步降低功耗。

LPC2138 通过功率控制寄存器 PCON 和外设功率控制寄存器 PCONP 进行功率控制。

1．功率控制寄存器 PCON

PCON 寄存器的功能如表 3-14 所列，置位 IDL 和 PD 两位中的一位，将会进入对应的节电模式，如果两位都置位，则进入掉电模式。

表 3-14　PCON 寄存器功能

位	功　能	功 能 说 明	复 位 值
0	IDL	空闲模式。该位置位时，处理器时钟停止，但外围功能保持工作状态。外设或外部中断源所产生的任何中断都会使处理器恢复运行	0
1	PD	掉电模式。该位置位时，振荡器和所有片内时钟都停止。外部中断所产生的唤醒条件可重新启动振荡器并清零 PD 位，处理器恢复运行	0
2	PDBOD	当 PD 为 1，PDBOD 为 0 时，掉电模式下掉电检测继续运行，以致它的复位使芯片退出掉电模式。当 PD 和 PDBOD 都为 1 时，掉电模式中 BOD 电路被禁止以节省功耗。当 PD 为 0 时，PDBOD 的状态对上述操作不会产生影响	
7:2	保留	未使用	NA

PCONP 寄存器允许将所选的外设功能关闭以实现节电的目的，这是通过关断特定外围模块的时钟源来实现的。看门狗定时器、GPIO、引脚连接模块和系统控制模块功能不

能被关闭。

某些外设，特别是包含模拟功能的外设，它们的操作无需时钟，但会消耗功率。这些外设包含独立的禁止控制位，可以通过它们来关闭电路以降低功耗。例如 AD 转换控制寄存器 ADCR 中含有独立的掉电位 PDN，在关闭 PCONP 中的 PCAD0 位之前，应该先将 ADCR 寄存器中的 PDN 清零。

2．外设功率控制寄存器 PCONP

PCONP 寄存器的功能如表 3-15 所列，PCONP 中每个位都控制一个相应的外设。

表 3-15　PCONP 寄存器功能

位	功　能	功 能 说 明	复 位 值
0	保留	未使用	0
1	PCTIM0	为 1 时，定时器 0 被激活；为 0 时，定时器 0 被关闭以实现节电	1
2	PCTIM1	为 1 时，定时器 1 被激活；为 0 时，定时器 1 被关闭以实现节电	1
3	PCURT0	为 1 时，UART 被激活；为 0 时，UART0 被关闭以实现节电	1
4	PCURT1	为 1 时，UART1 被激活；为 0 时，UART1 被关闭以实现节电	1
5	PCPWM0	为 1 时，PWM0 被激活；为 0 时，PWM0 被关闭以实现节电	1
6	保留	未使用	0
7	PCI^2C0	为 1 时，I^2C0 接口被激活；为 0 时，I^2C0 接口被关闭以实现节电	1
8	PCSPI0	为 1 时，SPI0 接口被激活；为 0 时，SPI0 接口被关闭以实现节电	1
9	PCRTC	为 1 时，RTC 被激活；为 0 时，RTC 被关闭以实现节电	1
10	PCSPI1	为 1 时，SSP 接口被激活；为 0 时，SSP 接口被关闭以实现节电	1
11	保留	用户软件写入 0 来实现节电	1
12	PCAD0	为 1 时，A-D0 被激活；为 0 时，A-D0 被关闭以实现节能；清零该位之前先应先清零 AD0CR 寄存器的 PDN 位；该位应当在置位 PDN 前被置位	1
18:13	保留	未使用	0
19	PCI^2C1	该位为 1 时，I^2C1 接口使能；为 0 时，I^2C1 接口被关闭以实现节电	1
31：20	保留	未使用	0

在进入空闲/掉电模式之前，应先将各个外设功能关闭，以降低功耗。若要使用外部中断来唤醒掉电，则需要开放外部中断，并设置好 INTWAKE 寄存器。

使用 LPC2138 功率控制时需要注意，芯片复位后，PCONP 的值设置成允许所有接口和外围功能（受 PCONP 控制）。除了对外围功能相关的寄存器进行配置外，用户应用程序不要访问 PCONP 寄存器，以免启动使用片内的任何外围功能。

在需要进行功率控制的系统中，只要将应用中用到的外围功能对应在 PCONP 寄存器中的位置 1，而寄存器中的其他位都必须清零。

3.2.4　复位

LPC2138 有两个复位源：RESET 引脚和看门狗复位。RESET 为施密特触发输入引脚，带有一个额外的干扰滤波器。任何复位源引起的芯片复位都会启动唤醒定时器，复位将保持有效直至外部复位撤除，振荡器开始正常运行。当计数达到一个固定个数的时钟时，片内电路完成其初始化。

复位干扰滤波器使处理器可以忽略非常短的外部复位脉冲，并决定 RESET 保证芯片复位所必须保持的最短时间。RESET 一旦有效，只有当晶振运行稳定并且 LPC2138 的 XTAL1 引脚上出现适当的信号时才能撤除。如果使用外部晶振，上电后 RESET 引脚上的信号必须保持 10ms。对于晶振已经稳定运行且 XTAL1 引脚上已出现稳定信号时出现的复位，RESET 引脚上的信号则只需要保持 300ns。

当内部复位撤除时，处理器从 0x00000000 地址开始运行，此处为从 BootBlock 映射的复位向量。此时所有的处理器状态和外设寄存器状态都恢复为默认值。

外部复位和内部看门狗复位有一些小的区别。外部复位使特定引脚的值被锁存以实现配置，内部看门狗复位则无此功能。外部复位时，对引脚 P1.20/TRACESYNC、P1.26/RTCK 的状态进行检测，以实现不同的目的。当复位后执行引导装载程序时，片内引导装载程序将对 P0.14 进行检测，以判断是否执行 ISP 操作。

LPC2138 的复位处理流程如图 3-8 所示。

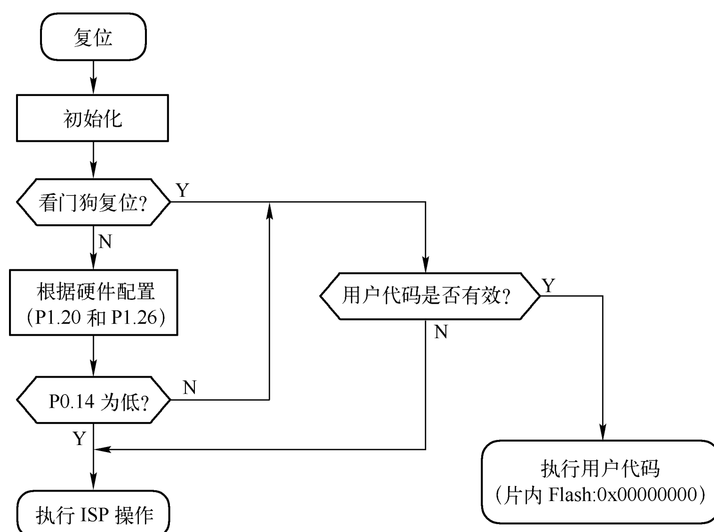


图 3-8　LPC2138 的复位处理流程

芯片复位可以发生在 Flash 编程或擦除操作过程中，此时，Flash 存储器会中断正在进行的操作，并使 CPU 复位延迟到内部 Flash 高电压降低后才完成。

3.2.5　唤醒定时器

唤醒定时器的用途是确保振荡器和芯片所需要的其他模拟功能在处理器开始执行指令之前能够正确工作。这在上电、所有类型的复位以及任何原因所导致上述功能关闭时非常重要。由于振荡器和其他功能在掉电模式下关闭，因此，处理器从掉电模式中唤醒时，必须使用唤醒定时器。

唤醒定时器通过检测晶振是否能可靠地开始代码的执行来对其进行监视。当给芯片上电或某个事件使芯片退出掉电模式时，振荡器需要一段时间来产生足够振幅的信号驱动时钟逻辑，所需时间取决于许多因素，包括 Vdd 电源的上升速率（上电时）、晶振的类型及其电气特性（如果使用石英晶振）、任何其他外部电路（例如电容）和振荡器在现有环境下

自身的特性等。

一旦检测到一个时钟，唤醒定时器即开始对 4096 个时钟计数，这段时间可使片内 Flash 进行初始化。当片内 Flash 初始化完毕时，如果外部复位已撤除，处理器就开始执行指令。当系统使用外部时钟源时，需要考虑的是振荡器的启动延时可能很短甚至没有。唤醒定时器的设计确保了芯片所需要的任何其他功能在程序运行之前都能够进行操作。

复位、外部中断和 RTC 中断都可使 LPC2138 退出掉电模式。如果一个中断使能产生唤醒并且所选中断事件出现，那么唤醒定时器将被启动。实际的中断（如果有的话）在唤醒定时器停止后产生，由向量中断控制器进行处理。

LPC2138 的引脚复用功能允许其他外设起作用，使芯片退出掉电模式。

要使芯片进入掉电模式并允许总线或引脚上的一个或多个事件能使其恢复正常操作，应对引脚的外部中断功能重新编程，选择中断合适的方式和极性以及掉电模式。唤醒时软件应恢复引脚复用的外围功能。

如果软件要使芯片退出掉电模式来响应多个引脚共用的同一个外部中断通道的事件，中断通道必须编程设定为低电平激活方式，因为只有在电平方式中，通道才能使逻辑“或”来唤醒芯片。这种方法的唯一缺陷，是振荡器重新启动会阻止 LPC2138 对总线或引脚上唤醒事件的捕获。空闲模式比掉电模式更适合于及时捕获和响应外部活动。

总之，LPC2138 唤醒定时器是根据晶振的情况来执行最短时间的复位，它使芯片在从掉电模式中唤醒或任何复位产生时激活。

3.3 外部中断输入

LPC2138 含有 4 个外部中断输入（作为可选的引脚功能）。外部中断输入可用于将处理器从掉电模式唤醒。

可将多个引脚同时连接同一路外部中断，此时，外部中断逻辑根据方式位和极性位的不同，分别进行如下处理：

- 在低有效电平激活方式中，选用外部中断功能的全部引脚的状态都连接到一个正逻辑“与”门。
- 在高有效电平激活方式中，选用外部中断功能的全部引脚的状态都连接到一个正逻辑“或”门。
- 在边沿激活方式中，使用 GPIO 端口号最低的引脚，与引脚的极性无关（在边沿激活方式中选择使用多个外部中断引脚被认为是编程出错）。

当多个外部中断引脚进行逻辑“或”时，可在中断服务程序中，通过 IO0PIN 和 IO1PIN 寄存器从 GPIO 端口读出引脚状态，来判断产生中断的引脚。

与外部中断相关的寄存器有 4 个：外部中断标志寄存器 EXTINT，其中包含中断标志；外部中断唤醒寄存器 INTWAKE，其中包含使能唤醒位，可使能独立的外部中断输入将处理器从掉电模式唤醒；外部中断方式寄存器 EXTMODE 和外部中断极性寄存器 EXTPOLAR，用来指定外部中断输入引脚所使用的电平或边沿激活方式。

1．外部中断标志寄存器 EXTINT

当一个引脚选择使用外部中断功能时，通过相应激活电平或边沿，将 EXTINT 寄存器中

的中断标志置位，并向 VIC 提出中断请求，如果引脚中断使能，将会产生中断。

向 EXTINT 寄存器的 EINT0～EINT3 位写入 1，可清除相应的外部中断标志。在电平激活方式下，只有在该引脚处于无效状态时，才能清除相应的中断标志。

一旦 EINT0～EINT3 中的一位被置位并开始执行相应的代码（处理唤醒和/或外部中断），必须将该位清零，否则，以后该 EINT 引脚所触发的事件将不能再被识别。例如，如果外部中断 0 引脚的低电平将系统从掉电模式唤醒，为了将来还能进入掉电模式，唤醒后的程序必须将 EINT0 位清零。如果 EINT0 位仍保持置位状态，则以后唤醒掉电模式的任何操作都将失败，外部中断也不例外。

EXTINT 寄存器的功能如表 3-16 所列。

表 3-16　EXTINT 寄存器功能

位	功　能	功 能 说 明	复 位 值
0	EINT0	电平激活方式下，如果引脚的 EINT0 功能被选用且引脚处于有效状态时，该位置位；边沿激活方式下，如果引脚的 EINT0 功能被选用且引脚上出现所选极性，该位置位 该位通过写入 1 清除，但电平激活方式下引脚处于有效状态的情况除外	0
1	EINT1	电平激活方式下，如果引脚的 EINT1 功能被选用且引脚处于有效状态时，该位置位；边沿激活方式下，如果引脚的 EINT1 功能被选用且引脚上出现所选极性，该位置位 该位通过写入 1 清除，但电平激活方式下引脚处于有效状态的情况除外	0
2	EINT2	电平激活方式下，如果引脚的 EINT2 功能被选用且引脚处于有效状态时，该位置位；边沿激活方式下，如果引脚的 EINT2 功能被选用且引脚上出现所选极性，该位置位 该位通过写入 1 清除，但电平激活方式下引脚处于有效状态的情况除外	0
3	EINT3	电平激活方式下，如果引脚的 EINT3 功能被选用且引脚处于有效状态时，该位置位；边沿激活方式下，如果引脚的 EINT3 功能被选用且引脚上出现所选极性，该位置位 该位通过写入 1 清除，但电平激活方式下引脚处于有效状态的情况除外	0
7:4	保留	未使用	NA

2．外部中断唤醒寄存器 INTWAKE

INTWAKE 寄存器中的使能位允许外部中断、BOD 中断或者 RTC 中断将处理器从掉电模式唤醒。相关的 EINTn 功能必须映射到引脚才能实现掉电唤醒，但中断并不需要为了实现唤醒操作而在向量中断控制器中被使能。这样做的好处是允许直接采用外部中断输入将处理器从掉电模式唤醒，而不产生中断（只是简单地恢复操作），或者在掉电模式下使能中断而不将处理器唤醒（这样，当应用中并不需要唤醒特性时，也不必关闭中断）。

INTWAKE 寄存器的功能如表 3-17 所列。

表 3-17　INTWAKE 寄存器功能

位	功　能	功 能 说 明	复 位 值
0	EXTWAKE0	该位为 1 时，使能 EINT0 将处理器从掉电模式唤醒	0
1	EXTWAKE1	该位为 1 时，使能 EINT1 将处理器从掉电模式唤醒	0
2	EXTWAKE2	该位为 1 时，使能 EINT2 将处理器从掉电模式唤醒	0
3	EXTWAKE3	该位为 1 时，使能 EINT3 将处理器从掉电模式唤醒	0
13:4	保留	未使用	NA

（续）

位	功　　能	功 能 说 明	复　位　值
14	BODWAKE	该位为 1 时，BOD 中断产生时可将处理器从掉电模式唤醒	0
15	RTCWAKE	该位为 1 时，RTC 中断产生时可将处理器从掉电模式唤醒	0

要使器件进入掉电模式并允许总线或引脚上的一个或多个事件能使其恢复正常操作，软件应该对引脚的外部中断功能重新编程。唤醒时软件应恢复引脚复用的外围功能。

上述所有总线或引脚活动都是低电平有效。如果希望以多个引脚共用的同一个 EINT 通道来使器件退出掉电模式，中断通道必须编程设定为低电平激活方式，因为，只有在电平方式下才能使用信号逻辑“或”来唤醒器件。

3．外部中断方式寄存器 EXTMODE

EXTMODE 寄存器用来选择每个 EINT 脚是电平触发还是边沿触发。只有通过引脚连接模块选择用作外部中断功能并已通过向量中断使能寄存器 VICIntEnable 使能的引脚才能产生外部中断（如果引脚选择用作其他功能，则可能产生其他功能的中断）。

EXTMODE 寄存器的功能如表 3-18 所列。

表 3-18　EXTMODE 寄存器功能

位	功　　能	功 能 说 明	复　位　值
0	EXTMODE0	该位为 0 时，EINT0 使用电平激活；该位为 1 时，EINT0 使用边沿激活	0
1	EXTMODE1	该位为 0 时，EINT1 使用电平激活；该位为 1 时，EINT1 使用边沿激活	0
2	EXTMODE2	该位为 0 时，EINT2 使用电平激活；该位为 1 时，EINT2 使用边沿激活	0
3	EXTMODE3	该位为 0 时，EINT3 使用电平激活；该位为 1 时，EINT3 使用边沿激活	0
7:4	保留	未使用	NA

4．外部中断极性寄存器 EXTPOLAR

在电平激活方式中，EXTPOLAR 寄存器用来选择相应引脚是高电平还是低电平有效；在边沿激活方式中，EXTPOLAR 寄存器用来选择引脚上升沿还是下降沿有效。只有通过引脚连接模块选择用作外部中断功能并已通过向量中断使能寄存器 VICIntEnable 使能的引脚才能产生外部中断（如果引脚选择用作其他功能，则可能产生其他功能的中断）。

EXTPOLAR 寄存器的功能如表 3-19 所列。

表 3-19　EXTPOLAR 寄存器功能

位	功　　能	功 能 说 明	复　位　值
0	EXTPOLAR0	该位为 0 时，EINT0 低电平或下降沿有效（由 EXTMODE0 决定）； 该位为 1 时，EINT0 高电平或上升沿有效（由 EXTMODE0 决定)	0
1	EXTPOLAR1	该位为 0 时，EINT1 低电平或下降沿有效（由 EXTMODE1 决定）； 该位为 1 时，EINT1 高电平或上升沿有效（由 EXTMODE1 决定)	0
2	EXTPOLAR2	该位为 0 时，EINT2 低电平或下降沿有效（由 EXTMODE2 决定）； 该位为 1 时，EINT2 高电平或上升沿有效（由 EXTMODE2 决定)	0
3	EXTPOLAR3	该位为 0 时，EINT3 低电平或下降沿有效（由 EXTMODE3 决定）； 该位为 1 时，EINT3 高电平或上升沿有效（由 EXTMODE3 决定）	0
7:4	保留	未使用	NA

把相应引脚设置为外部中断功能时，引脚为输入模式，由于没有内部上拉电阻，用户需要外接一个上拉电阻，确保引脚不会悬空。

下面的语句可将 EINT0 设置为电平中断。

```
PINSEL1 = 0x00000001;    // 设定 P0.16 连接外部中断 0
EXTMODE = 0x00;          // 外部中断 0 为电平触发
EXTPOLAR = 0x00;         // 低电平触发
或
EXTPOLAR = 0x01;         // 高电平触发
```

下面的语句可将 EINT0 设置为上升/下降沿中断。

```
PINSEL1 =0x00000001;     // 设定 P0.16 连接外部中断 0
EXTMODE = 0x01;          // 外部中断 0 为边沿触发
EXTPOLAR = 0x00;         // 下降沿触发
或
EXTPOLAR = 0x01;         // 上升沿触发
```

下面的语句将清除所有外部中断。

```
EXTINT = 0x0F;
```

3.4 向量中断控制器 VIC

向量中断控制器 VIC 接收所有的中断请求输入，并将它们分为 3 类：快中断请求 FIQ、向量中断请求 IRQ 和非向量中断请求 IRQ。可编程实现不同外设中断优先级的动态分配和调整。

快中断 FIQ 是优先级别最高的中断。为了确保 FIQ 响应的最短延时，实际应用中，一般只设置一个 FIQ 中断类型。如果有一个以上的中断源被设置为 FIQ 中断，VIC 将对中断请求进行“或”操作，最终向 ARM 内核产生一个 FIQ 信号。这时应在 FIQ 中断服务程序中先读出 VIC 的 FIQ 中断状态字，判断出真正发生的中断源之后，才能处理对应的中断。

当只有一个中断请求被分配为 FIQ 时，可实现最短的 FIQ 等待时间，此时 FIQ 服务程序可以直接从 FIQ 向量地址开始执行。如果分配给 FIQ 的中断请求多于一个，则 FIQ 服务程序需要读取 FIQ 状态寄存器，以识别产生中断请求的 FIQ 中断源。

向量 IRQ 的优先级为中等。该级别可分配 32 个中断请求中的 16 个。32 个中断请求中的任意一个都可分配到 16 个向量 IRQ slot 中的任意一个，其中 slot0 最高，slot15 最低。

非向量 IRQ 的优先级最低。

VIC 将所有向量 IRQ 和非向量 IRQ“相或”后向 ARM 处理器产生 IRQ 信号。IRQ 服务程序可通过读取 VIC 的一个寄存器立即启动并跳转到相应地址。如果存在任何一个未决的向量 IRQ，VIC 将提供最高优先级请求 IRQ 服务程序的地址，否则提供由所有非向量 IRQ 共用的默认程序地址。

3.4.1 VIC 寄存器

VIC 所包含多个寄存器，所有寄存器都为字寄存器。不支持字节和半字的读和写操作。

下面按照 VIC 逻辑中的使用顺序对 VIC 寄存器进行描述，该顺序为：从与中断请求输入最密切的寄存器到由软件使用的最抽象的寄存器。

1．软件中断寄存器 VICSoftInt

在执行任何逻辑之前，将该寄存器的内容与 32 个不同外设的中断请求相或。VICSoftInt 寄存器的功能如表 3−20 所列。

表 3−20　VICSoftInt 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	1：强制产生与该位相关的中断请求 0：不强制产生中断请求。向 VICSoftInt 写入 0 无效	读/写	0

2．软件中断清零寄存器 VICSoftIntC1

该寄存器在不需读取软件中断寄存器的情况下，可用软件清零软件中断寄存器中的一个。VICSoftIntC1 寄存器的功能如表 3−21 所列。

表 3−21　VICSoftIntC1 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	1：写入 1 清零软件中断寄存器的相应位，并删除强制的中断请求 0：写入 0 不会影响 VICSoftInt 中的相应位	只写	0

3．所有中断状态寄存器 VICRawIntr

该寄存器读取所有 32 个中断请求和软件中断的状态，不管中断是否使能或分类。VICRawIntr 寄存器的功能如表 3−22 所列。

表 3−22　VICRawIntr 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	1：对应位的中断请求或软件中断声明 0：对应位的中断请求或软件中断未声明	只读	0

4．中断使能寄存器 VICIntEnable

该寄存器使能分配为 FIQ 或 IRQ 的中断请求或软件中断。VICIntEnable 寄存器的功能如表 3−23 所列。

表 3−23　VICIntEnable 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	读取该寄存器时，1 表示中断请求使能为 FIQ 或 IRQ 写入 1 使能中断请求或软件中断分配为 FIQ 或 IRQ，写入 0 无效	读/写	0

5．中断使能清零寄存器 VICIntEnC1

该寄存器在不需要读取中断使能寄存器的情况下，可用软件清零其中的一个或多个位。VICIntEnC1 寄存器的功能如表 3−24 所列。

表 3-24　VICIntEnC1 寄存器功能

位	功 能 说 明	访　　问	复 位 值
31:0	1：写入 1 清零中断使能寄存器中的对应位并禁止对应的中断请求 0：写入 0 不影响中断使能寄存器中的位	只写	0

6．中断选择寄存器 VICIntSelect

该寄存器将 32 个中断请求分别分配为 FIQ 或 IRQ。VICIntSelect 寄存器的功能如表 3-25 所列。

表 3-25　VICIntSelect 寄存器功能

位	功 能 说 明	访　　问	复 位 值
31:0	1：对应的中断请求分配为 FIQ 0：对应的中断请求分配为 IRQ	读/写	0

7．IRQ 状态寄存器 VICIRQStatus

该寄存器读取使能并分配为 IRQ 的中断请求状态，不区分向量和非向量 IRQ。VICIRQStatus 寄存器的功能如表 3-26 所列。

表 3-26　VICIRQStatus 寄存器功能

位	功 能 说 明	访　　问	复 位 值
31:0	1：对应位的中断请求使能，并分配为 IRQ 且声明	只读	0

8．FIQ 状态寄存器 VICFIQStatus

该寄存器读取使能并分配为 FIQ 的中断请求状态。如果有超过一个请求分配为 FIQ，FIQ 服务程序可读取该寄存器来确定是哪个请求被激活。VICFIQStatus 寄存器的功能如表 3-27 所列。

表 3-27　VICFIQStatus 寄存器功能

位	功 能 说 明	访　　问	复 位 值
31:0	1：对应位的中断请求使能，并分配为 FIQ 且声明	只读	0

9．向量控制寄存器 VICVectCnt10～VICVectCntl15

每个寄存器控制 16 个向量 IRQ slot 中的一个。slot0 最高，slot15 最低。在 VICVectCnt1 寄存器中禁止一个向量 IRQ slot 不会禁止中断本身，中断只是变为非向量的形式。VICVectCnt10～15 寄存器的功能如表 3-28 所列。

表 3-28　VICVectCntl0~15 寄存器功能

位	功 能 说 明	访　　问	复 位 值
5	1：向量 IRQ 使能。当分配的中断请求或软件中断使能，被分配为 IRQ 并声明时，可产生一个唯一的 ISR 地址	读/写	0
4:0	分配给此向量 IRQ slot 的中断请求或软件中断的编号。不要将相同的中断编号分配给多于一个使能的向量 IRQ slot，但如果这样做了，当中断请求或软件中断使能，被分配为 IRQ 并声明时，会使用最低编号的 slot		

10．向量地址寄存器 VICVectAddr0～VICVectAddr15

这些寄存器保存 16 个向量 IRQ slot 中断服务程序的地址，VICVectAddr0～15 寄存器的功能如表 3-29 所列。

表 3-29　VICVectAddr0～15 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	当一个或多个中断请求或软件分配为向量 IRQ slot 的中断请求使能，且分配为 IRQ 并声明时，IRQ 服务程序读取向量地址寄存器（VICVectAddr）时会得到最高 slot 寄存器的值	读/写	0

11．默认向量地址寄存器 VICDefVectAddr

该寄存器保存非向量 IRQ 中断服务程序的地址，VICDefVectAddr 寄存器的功能如表 3-30 所列。

表 3-30　VICDefVectAddr 寄存器功能

位	功能说明	访　问	复 位 值
31:0	当一个 IRQ 服务程序读取向量地址寄存器（VICVectAddr），且没有 IRQ slot 响应时，则返回该寄存器中的地址	读/写	0

12．向量地址寄存器 VICVectAddr

当发生一个 IRQ 中断时，IRQ 服务程序可读取该寄存器并跳转到读出的地址，VICVectAddr 寄存器的功能如表 3-31 所列。

表 3-31　VICVectAddr 寄存器功能

位	功 能 说 明	访　问	复 位 值
31:0	当任何分配给向量 IRQ slot 的中断请求或软件中断使能，且分配为 IRQ 并声明时，读取该寄存器将返回最高 slot（最低编号）在向量地址寄存器中的地址，否则返回默认向量地址寄存器中的地址 由于写入该寄存器的设置值并不能在将来读出，因此，在 ISR 快结束时应该对此寄存器执行写操作（通常写入 0），以便更新优先级硬件	读/写	0

13．保护使能寄存器 VICProtection

运行在用户模式下的软件使用该寄存器来控制对 VIC 寄存器的访问，VICProtection 寄存器的功能如表 3-32 所列。

表 3-32　VICProtection 寄存器功能

位	功 能 说 明	访　问	复 位 值
0	1：VIC 寄存器只能在特权模式下访问 0：VIC 寄存器可在用户模式或特权模式下访问	读/写	0

3.4.2　中断源

表 3-33 所列为 LPC2138 各个外设功能的中断源。每个外设都有一条中断线连接到向量中断控制器，但有些可能拥有几个内部中断标志。单个中断标志也可能代表多个中断源。

表 3-33　各个外围功能的中断源

模　　块	标　　志	VIC 通道号
WDT	看门狗中断（WDINT）	0
—	保留给软件中断使用	1
ARM 内核	EmbeddedICE, DbgCommRx	2
ARM 内核	EmbeddedICE, DbgCommTx	3
定时器 0	匹配 0～3（MR0, MR1, MR2, MR3） 捕获 0～3（CR0, CR1, CR2, CR3）	4
定时器 1	匹配 0～3（MR0, MR1, MR2, MR3） 捕获 0～3（CR0, CR1, CR2, CR3）	5
UART0	Rx 线状态（RLS） 发送保持寄存器空（THRE） Rx 数据可用（RDA） 字符超时指示（CTI）	6
UART1	Rx 线状态（RLS） 发送保持寄存器空（THRE） Rx 数据可用（RDA） 字符超时指示（CTI） 调制解调器状态中断（MSI）	7
PWM	匹配 0～6（MR0, MR1, MR2, MR3, MR4, MR5, MR6）	8
I^2C0	SI（状态改变）	9
SPI0	SPI 中断标志（SPIF） 模式错误（MODF）	10
SPI1(SSP)	Tx FIFO 至少一半为空（TXRIS） Rx FIFO 至少一半为满（RXRIS） 接收超时（RTRIS） 接收溢出（RORRIS）	11
PLL	PLL 锁定（PLOCK)	12
RTC	计数器增加（RTCCIF） 报警（RTCALF）	13
系统控制	外部中断 0（EINT0）	14
	外部中断 1（EINT1）	15
	外部中断 2（EINT2）	16
	外部中断 3（EINT3）	17
AD0	A-D 转换器 0	18
I^2C1	SI（状态改变）	19
BOD	掉电检测	20
AD1	A-DC 转换器 1	21

中断源与 VIC 的连接如图 3-9 所示。

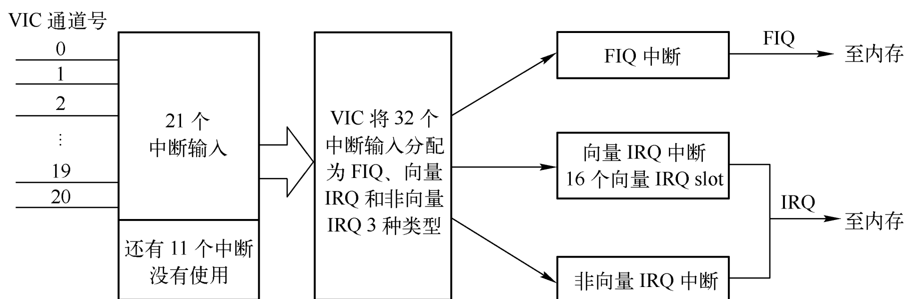


图 3-9　中断源与 VIC 的连接图

3.4.3　VIC 使用注意事项

VIC 使用中需要注意以下几点：

1）如果在片内 RAM 中运行用户代码并且应用程序需要使用中断，那么必须将异常向量重新映射到片内 Flash 地址 0x00000000。这样做是因为所有的异常向量都位于地址 0x00000000 及以上。通过配置 MEMMAP 寄存器（位于系统控制模块当中），为用户 RAM 模式来实现这一点。另外，用户程序代码在编译链接时，应使用异常向量表装载到 0x40000000。

2）虽然可以选择多个中断源（通过 VICIntSelect）来产生 FIQ 请求，但是只有一个专门的中断服务程序来响应所有可用的 FIQ 请求。因此，如果分配为 FIQ 的中断多于一个，FIQ 中断服务程序就必须读取 VICFIQStatus 的内容来决定如何处理中断请求。建议只将一个中断分配为 FIQ，因为多个 FIQ 中断源会增加中断延迟。

3）中断服务程序执行完毕后，对外设中断标志的清零将会对 VIC 寄存器（VICRawlntr、VICFIQStatus、VICIRQStatus）当中的对应位产生影响。另外，为了能够服务下次中断，必须在中断返回之前，对 VICVectAddr 寄存器执行写操作。该写操作将清零内部中断优先级硬件当中对应的中断标志。

4）为了禁止 VIC 中断，必须清零 VICIntEnClr 寄存器中的对应位，该操作使 VICIntEnable 寄存器的对应位清零。这同样可应用于 VICSoftInt 和 VICSoftIntClear，VICSoftIntClear 将会使 VICSoftInt 中的对应位清零。例如，如果 VICSoftInt=0x0000 0005，并且需要清零 bit0，那么执行 ICSoftIntClear=0x0000 0001 即可实现该操作。

5）如果看门狗只在溢出或无效喂狗时产生中断，那么无法清除中断。唯一的方法是通过 VICIntEnClr 禁止 VIC 中断，来实现中断返回。

下面给出一个多中断设置的例子。

假设 UART0 和 SPI0 产生中断请求，它们被分配为向量 IRQ（UART0 的优先级高于 SPI0），而 UART1 和 I^2C 产生非向量 IRQ，下面就是一种 VIC 可能的设置：

VICIntSelect　= 0x0000 0000　(SPI0, I^2C, UART1 和 UART0 为 IRQ => bit10, bit9, bit7 和 bit6=0)

VICIntEnable　= 0x0000 06C0　(SPI0, I^2C, UART1 和 UART0 中断使能 => bit10, bit9, bit 7 和 bit6=1)

VICDefVectAddr　= 0x…　(保存服务非向量 IRQ 的程序地址，即 UART1 和 I^2C 的起始地址)

VICVectAddr0　　= 0x…　　　　　(保存 UART0 IRQ 服务程序的起始地址)
VICVectAddr1　　= 0x…　　　　　(保存 SPI0 IRQ 服务程序的起始地址)
VICVectCntl0　　= 0x0000 0026　　(VIC 通道号为 6（UART0）的中断源使能为优先级 0（最高级）)
VICVectCntl1　　= 0x0000 002A　　(VIC 通道号为 10（SPI0）中断源使能为优先级 1)

在任何 IRQ 请求（SPI0、I^2C、UART0 或 UART1）产生之后，处理器跳转到地址 0x00000018 执行代码。对于向量和非向量 IRQ，可在地址 0x18 放入下面指令：

```
LDR    PC，[PC, # -0xFF0]
```

该指令将 VICVectAddr 寄存器中保存的地址装入 PC 中。

一旦产生 UART0 请求，VICVectAddr 和 VICVectAddr0 相同。如果产生 SPI0 请求，中断服务程序的地址保存在 VICVectAddr1 中。如果 UART0 和 SPI0 都没有产生 IRQ 请求，而 UART1 和/或 I^2C 产生请求，那么 VICVectAddr 的内容与 VICDefVectAddr 相同。

3.5　引脚功能配置

LPC2138 的引脚大多是功能复用的，可以通过引脚连接模块在多个功能之间进行选择。引脚连接模块通过配置引脚选择寄存器控制片内多路开关来实现引脚与多种片内功能外设之间的连接，可实现独立的引脚配置。

片内功能外设在激活以及任何相关中断使能之前，必须连接到适当的引脚。任何使能的外设功能如果没有连接到相关引脚，都是无效的。

当引脚选择了一种功能时，其他功能无效。

3.5.1　引脚选择寄存器

引脚连接模块包括三个引脚选择寄存器：PINSEL0、PINSEL1 和 PINSEL2，芯片复位时这些寄存器会自动设置为默认值，所以复位后芯片引脚的功能是确定的。下面分别介绍三个引脚选择寄存器的功能。

1．引脚选择寄存器 0 PINSEL0

PINSEL0 寄存器的功能如表 3-34 所列。

表 3-34　PINSEL0 寄存器功能

位	引　脚　名	00	01	10	11	复　位　值
1：0	P0.0	GPIO	TxD0(UART0)	PWM1	保留	00
3：2	P0.1	GPIO	RxD0(UART0)	PWM3	EINT0	00
5：4	P0.2	GPIO	SCL0(I^2C0)	捕获 0.0 (TIMER0)	保留	00
7：6	P0.3	GPIO	SDA0(I^2C0)	匹配 0.0(TIMER0)	EINT1	00
9：8	P0.4	GPIO	SCK0(SPI0)	捕获 0.1(TIMER0)	AD0.6	00
11：10	P0.5	GPIO	MISO0(SPI0)	匹配 0.1(TIMER0)	AD0.7	00
13：12	P0.6	GPIO	MOSI0(SPI0)	捕获 0.2(TIMER0)	AD1.0	00
15：14	P0.7	GPIO	SSEL0(SPI0)	PWM2	EINT2	00

（续）

位	引　脚　名	00	01	10	11	复　位　值
17：16	P0.8	GPIO	TxD1(UART1)	PWM4	AD1.1	00
19：18	P0.9	GPIO	RxD1(UART1)	PWM6	EINT3	00
21：20	P0.10	GPIO	RTS(UART1)	捕获 1.0(TIMER1)	AD1.2	00
23：22	P0.11	GPIO	CTS(UART1)	捕获 1.1(TIMER1)	SCL1(I^2C1)	00
25：24	P0.12	GPIO	DSR(UART1)	匹配 1.0(TIMER1)	AD1.3	00
27：26	P0.13	GPIO	DTR(UART1)	匹配 1.1(TIMER1)	AD1.4	00
29：28	P0.14	GPIO	CD(UART1)	EINT1	SDA1(I^2C1)	00
31：30	P0.15	GPIO	RI(UART1)	EINT2	AD1.5	00

2．引脚选择寄存器 1 PINSEL1

PINSEL1 寄存器的功能如表 3-35 所列。

表 3-35　PINSEL1 寄存器功能

位	引　脚　名	00	01	10	11	复　位　值
1：0	P0.16	GPIO	EINT0	匹配 0.2(TIMER0)	捕获 0.2(TIMER0)	00
3：2	P0.17	GPIO	捕获 1.2(TIMER1)	SCK(SSP)	匹配 1.2(TIMER1)	00
5：4	P0.18	GPIO	捕获 1.3(TIMER1)	MISO(SSP)	匹配 1.3(TIMER1)	00
7：6	P0.19	GPIO	匹配 1.2(TIMER1)	MOSI(SSP)	匹配 1.3(TIMER1)	00
9：8	P0.20	GPIO	匹配 1.3(TIMER1)	SSEL(SSP)	EINT3	00
11：10	P0.21	GPIO	PWM5	AD1.6	捕获 1.3(TIMER1)	00
13：12	P0.22	GPIO	AD1.7	捕获 0.0(TIMER0)	匹配 0.0(TIMER0)	00
15：14	P0.23	GPIO	保留	保留	保留	00
17：16	P0.24	保留	保留	保留	保留	00
19：18	P0.25	GPIO	AD0.4	Aout（DAC）	保留	00
21：20	P0.26	保留	保留	保留	保留	00
23：22	P0.27	GPIO	AD0.0	捕获 0.1(TIMER0)	匹配 0.1(TIMER0)	00
25：24	P0.28	GPIO	AD0.1	捕获 0.2(TIMER0)	匹配 0.2(TIMER0)	00
27：26	P0.29	GPIO	AD0.2	捕获 0.3(TIMER0)	匹配 0.3(TIMER0)	00
29：28	P0.30	GPIO	AD0.3	EINT3	捕获 0.0(TIMER0)	00
31：30	P0.31	GPIO	保留	保留	保留	00

3．引脚选择寄存器 2 PINSEL2

PINSEL2 寄存器的功能如表 3-36 所列。

表 3-36　PINSEL2 寄存器功能

位	功 能 说 明	复　位　值
1：0	保留	00
2	该位为 0 时，P1[31:26]用作 GPIO 引脚 该位为 1 时，P1[31:26]用作一个调试端口	P1.26/ RTCK

（续）

位	功能说明	复位值
3	该位为 0 时，P1[25:16]用作 GPIO 引脚 该位为 1 时，P1[25:16]用作一个跟踪端口	P1.20/ TRACESYNC
31:4	保留	NA

3.5.2 引脚配置示例

选择引脚功能时，应先设置 PINSEL0、PINSEL1 或 PINSEL2 的值，让相应引脚连接到特定的片内外设，然后再设置特定外设相应的控制寄存器，这样才能使用特定的功能。LPC2138 芯片引脚往往具有 1 个以上的功能，而同一时刻只能使用 1 个功能。比如 P0.0 引脚可以用作 GPIO，也可以用作 UART0 的 TxD0，还可以用作脉宽调制器输出 PWM1，具体引脚功能视实际需要而定。

例如，将 P0.8、P0.9 设置为 TxD1、RxD1 功能，可以采用如下语句：

```
PINSEL0 = 0x00050000;
或 PINSEL0 = 0x05 << 16;
```

以上两条语句等价，第二条语句中 0x05<<16 是把 0101 左移 16 位，比第一条语句直观。

采用上面的语句写法将引脚 P0.8、P0.9 连接到 UART1，可能会改变其他引脚的功能设置，为了不更改原先的引脚功能设置，可以采取“读取－修改－回写”的方式进行，即先读取寄存器值，然后进行逻辑“与”、“或”操作，再回写到寄存器中。为此可将上面的语句改为

```
PINSEL0 = ( PINSEL0 & 0xFFF0FFFF ) | (0x05 << 16);
```

3.6 系统控制应用举例

3.6.1 存储器映射

LPC2138 的存储器映射控制是从处理 ARM 异常必需的 3 个数据源（Flash 异常向量、Sram 异常向量和 Boot Block 异常向量，每个异常向量为 64B）中选择一个使用。芯片复位时 MEMMAP=0，启动 Boot 装载程序，Boot 装载程序检查 P0.14 引脚的状态和用户的异常向量表，判断是进入 ISP 还是启动用户程序。若启动用户程序，则自动设置 MEMMAP=1，从片内 Flash 启动。若用户程序需要随时更改异常向量表，可以将 64B 的异常向量表复制到片内 RAM 从 0x40000000 地址开始的区域，从而可以对向量表进行修改。然后设置 MEMMAP=2 进行存储器重映射，使保存在 0x40000000 地址处的异常向量表映射到 0x00000000 地址上。

后面几个实例将采用如图 3-10 所示的 Proteus 仿真电路。

例 3-1 存储器重映射。

本例程序测试用户 RAM 模式和 Flash 模式下的存储器映射。默认从片内 Flash 启动用户程序，先将 64B 的异常向量表复制到片内 RAM 从 0x40000000 地址开始的区域，然后设置

MEMMAP=2 进行存储器重映射，使保存在 0x40000000 地址处的异常向量表映射到 0x00000000 地址上。这时在用户 RAM 区的 0x40000000～0x4000003F 处填入 0x55，可以发现在 0x00000000～0x0000003F 处的内容也变成了 0x55。

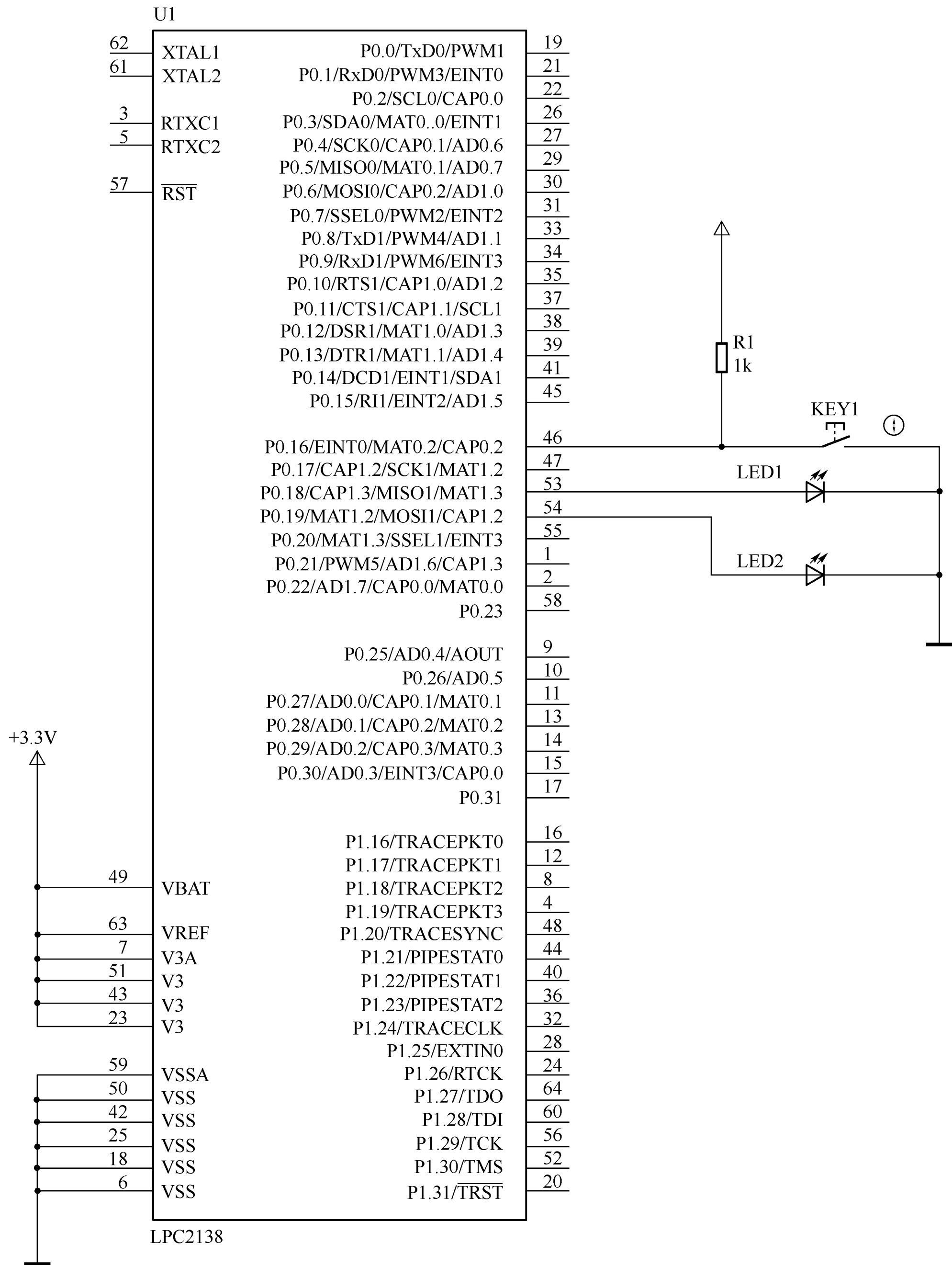


图 3-10　Proteus 仿真电路

需要注意的是，Proteus 中 ARM7 模型不能进行存储器重映射仿真，本例只有在实际硬件电路板上运行，才能够观察到正确的结果。

主程序文件 main.c 如下：

```
#include "config.h"
#define VECTOR_RAM ((unsigned char *)0x40000000)
/****************************************************************************************
** 函数名称：main()
** 函数功能：将存储器重映射到片内 RAM(0x40000000~0x4000003F)，
**           然后更改(0x40000000~0x4000003F)的内容
** 调试说明：先观察 0x40000000~0x4000003F 地址上的数据有何变化
**           执行 MAMMAP=2 后，看 0x00000000~0x0000003F 上的数据和
**           0x40000000～0x4000003F 的是否相同
****************************************************************************************/
int main (void)
{    uint8 i;
     uint8 *addr,*cp1,*cp2;
     cp1 = (unsigned char *)__iar_program_start;   // 向量表从起始标号处开始
     cp2 = VECTOR_RAM;                             // 内部 RAM 起始地址
     for(i=0; i<64; i++)
     {
          *cp2++ = *cp1++;               // 从 Flash 中拷贝向量表，16 个字
     }                                   // 注意观察 0x40000000～0x4000003F 的值

     MEMMAP = 0x02;                      // 异常向量从静态 RAM 重映射(用户 RAM 模式)
     addr = VECTOR_RAM;
     for (i=0; i<64; i++)
     {
          *addr = 0x55;                  // 用 0x55 填充 0x40000000~0x4000003F
          addr++;
     }                                   // 注意观察 0x00000000～0x0000003F 的值

     while (1);
}
```

3.6.2 锁相环

启动代码中包含一个用于对目标板进行初始化的文件 Target.c，可以对系统存储器、锁相环等进行设定，用户可以根据不同需要进行修改。

Target.c 文件如下：

```
#define IN_TARGET
#include "config.h"

/****************************************************************************************
** 函数名称: IRQ_Handler
** 功能描述: 中断异常处理程序，用户可根据需要进行修改
****************************************************************************************/
 __irq void IRQ_Handler (void)
```

```
{
   void (*interrupt_function)();
   unsigned int vector;
   vector = VICVectAddr;                   // 取得中断向量地址
   interrupt_function = (void(*)())vector;
   (*interrupt_function)();                // 跳转到中断函数
}

/*********************************************************************************
** 函数名称: FIQ_Exception
** 功能描述: 快速中断异常处理程序，用户根据需要自己改变程序
*********************************************************************************/
void FIQ_Exception(void)
{
     while(1);                             // 这一句替换为自己的代码
}

/*********************************************************************************
** 函数名称: TargetInit
** 功能描述: 目标板初始化代码，在需要的地方调用，根据需要改变
*********************************************************************************/
void TargetInit(void)
{
     /* 添加自己的代码 */
}

/*********************************************************************************
** 函数名称: TargetResetInit
** 功能描述: 调用 main 函数前目标板初始化代码，根据需要改变，不能删除
*********************************************************************************/
void TargetResetInit(void)
{
/* 设置存储器映射方式 */
#ifdef DEBUG_RAM                           // 如果在片内 RAM 调试
      MEMMAP = 0x02;                       // 映射到片内 RAM
#endif
#ifdef DEBUG_FLASH                         // 如果在片内 FLASH 调试
      MEMMAP = 0x01;                       // 映射到片内 FLASH
#endif
#ifdef IN_CHIP                             // 如果在片内 FLASH 发布
      MEMMAP = 0x01;                       // 映射到片内 FLASH
#endif

/* 设置系统各部分时钟 */
#ifdef PLL                                 // 如果需要设置 PLL
     PLLCON = 1;                           // 设置激活但未连接 PLL
```

```
#if ((Fcclk / 4) / Fpclk) == 1          // 此值由系统硬件决定
    VPBDIV = 0;
#endif
#if ((Fcclk / 4) / Fpclk) == 2          // 此值由系统硬件决定
    VPBDIV = 2;
#endif
#if ((Fcclk / 4) / Fpclk) == 4          // 此值由系统硬件决定
    VPBDIV = 1;
#endif

/* 设定 PLL 的乘因子 M 和除因子 P 的值 */
#if (Fcco / Fcclk) == 2
    PLLCFG = ((Fcclk / Fosc) - 1) | (0 << 5);
#endif
#if (Fcco / Fcclk) == 4
    PLLCFG = ((Fcclk / Fosc) - 1) | (1 << 5);
#endif
#if (Fcco / Fcclk) == 8
    PLLCFG = ((Fcclk / Fosc) - 1) | (2 << 5);
#endif
#if (Fcco / Fcclk) == 16
    PLLCFG = ((Fcclk / Fosc) - 1) | (3 << 5);
#endif
    PLLFEED = 0xaa;                         // 发送 PLL 馈送序列，执行设定 PLL 的动作
    PLLFEED = 0x55;
    while((PLLSTAT & (1 << 10)) == 0);      // 等待 PLL 锁定
    PLLCON = 3;                             // 设置激活并连接 PLL
    PLLFEED = 0xaa;                         // 发送 PLL 馈送序列，执行激活和连接动作
    PLLFEED = 0x55;
#endif

/* 设置存储器加速模块 */
    MAMCR = 0;                              // 禁止 MAM 功能
#if Fcclk < 20000000
    MAMTIM = 1;                             // 系统时钟低于 20MHz，建议设置为 1
#else
#if Fcclk < 40000000
    MAMTIM = 2;                             // 系统时钟在 20～40MHz 之间，建议设置为 2
#else
    MAMTIM = 3;                             // 系统时钟高于 40MHz，建议设置为 3
#endif
#endif
    MAMCR = 2;                              // 使能 MAM 功能

/* 初始化 VIC */
    VICIntEnClear = 0xffffffff;             // 清零所有中断
```

```
    VICVectAddr = 0;                    // 默认向量地址寄存器
    VICIntSelect = 0;                   // 0：中断请求分配为 IRQ, 1：中断请求分配为 FIQ

    /* 添加自己的代码 */

}
```

LPC2138 外接晶振频率范围为 1～30MHz，通过片内锁相环功能可实现最大为 60MHz 的 CPU 操作频率。正确设置锁相环的操作方法如下：

- 设置 PLLCON 寄存器，使能 PLL。
- 设置 PLLCFG 寄存器，选择倍频系数 M 和分频系数 P 的值。
- 发送 PLL 馈送序列。
- 等待 PLL 锁定（查询等待或中断方式）。
- 设置 PLLCON 寄存器，使能和连接 PLL。
- 再次发送 PLL 馈送序列。
- 禁止 PLL 中断（若采用中断方式的话）。

例 3-2 采用中断方式设置锁相环。

通常在进入主程序之前，通过启动代码的 Target.c 文件来进行锁相环设置，Target.c 文件中采用查询方式等待 PLL 锁定后，再使能和连接 PLL。本例不通过启动代码而是直接在主程序中进行锁相环设置，并以 IRQ 中断方式来使能和连接 PLL。程序运行后可以看到，PLL 锁定之前，LED1 以较慢速度闪烁，PLL 锁定之后，LED1 以较快速度闪烁。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>

#define LED1 1 << 18;        // P0.18 控制 LED1

/****************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时。
** 入口参数：dly 延时参数，值越大，延时越久
****************************************************************************/
void DelayNS (uint32 dly)
{     uint32 i;
      for ( ; dly>0; dly--)
      for (i=0; i<50000; i++);
}

/****************************************************************************
** 函数名称：PLL_INT()
** 函数功能：在中断中使能和连接 PLL
****************************************************************************/
```

```
void PLL_INT (void)
{	PLLCON = 3;				// 使能和连接 PLL
	PLLFEED = 0xAA;				// 发送 PLL 馈送序列
	PLLFEED = 0x55;
	VICIntEnClr = 1 << 12;			// 禁止 PLL 中断
	VICVectAddr = 0x00;			// 向量中断处理结束
}

/****************************************************************************************
** 函数名称：main()
** 函数功能：等待 PLL 中断，更改 PLL 设置
****************************************************************************************/
int main (void)
{	uint32 i;
	PINSEL0 = 0x00000000;			// 设置引脚连接 GPIO
	IO0DIR = LED1;				// 设置 LED1 控制口为输出，其余输入
	for (i=0; i<5; i++)			// LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
	{	IO0SET = LED1;
		DelayNS(5);
		IO0CLR = LED1;
		DelayNS(5);
	}

	PLLCON = 1;				// 使能 PLL
	PLLCFG = 0x24;				// 设置 M 为 5，P 为 2
	PLLFEED = 0xAA;				// 发送 PLL 馈送序列
	PLLFEED = 0x55;
	__enable_irq();				// 使能 IRQ 中断

	/* 设置 PLL 中断 */
	VICIntSelect = 0x00000000;		// 设置所有的通道为 IRQ 中断
	VICVectCntl0 = 0x20 | 12;		// PLL_INT 分配到 IRQ slot0，即最高优先级
	VICVectAddr0 = (uint32)PLL_INT;		// 设置 PLL_INT 向量地址
	VICIntEnable = 1 << 12;			// 使能 PLL_INT 中断

	for (i=0; i<5; i++)			// LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
	{	IO0SET = LED1;
		DelayNS(5);
		IO0CLR = LED1;
		DelayNS(5);
	}
	while (1);
}
```

3.6.3 存储器加速模块

LPC2138 的存储器加速模块可以提高 CPU 对 Flash 的访问速度。Target.c 文件中有一段设置存储器加速模块 MAM 的代码，其中 MAMTIM 的值越大，速度越快。

例3-3 存储器加速模块功能验证。

进入主程序之前，已经在启动代码中对存储器加速模块 MAM 进行了设置，在 main()函数中，采用语句 MAMCR = 0 将 MAM 关闭。程序全速运行后可以看到，LED1 开始以较快速度闪烁（MAM 开启），然后 LED1 以较慢速度闪烁（MAM 关闭），验证了 MAM 的加速作用。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define LED1 1 << 18;                  // P0.18  控制 LED1

/****************************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时
** 入口参数：dly  延时参数，值越大，延时越久
****************************************************************************************/
void DelayNS (uint32 dly)
{       uint32 i;
        for ( ; dly>0; dly--)
        for (i=0; i<50000; i++);
}

/****************************************************************************************
** 函数名称：main()
** 函数功能：验证存储器加速模块 MAM 对程序运行速度的影响
****************************************************************************************/
int main (void)
{       uint32 i;
        PINSEL1 = 0x00000001;      //  P0.16  连接 EINT0
        IO0DIR = LED1;             //  设置 LED1 控制口为输出，其余输入
        for (i=0; i<5; i++)        //  LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
        {       IO0SET = LED1;
                DelayNS(5);
                IO0CLR = LED1;
                DelayNS(5);
        }

        DelayNS(10);
/*  关闭 MAM */
        MAMCR = 0;                 //  关闭 MAM
```

```
    /* 关闭 MAM 后 LED1 闪烁 5 次，注意闪烁频率 */
        for (i=0; i<5; i++)         // LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
        {   IO0SET = LED1;
            DelayNS(5);
            IO0CLR = LED1;
            DelayNS(5);
        }
        while (1);
    }
```

3.7 VIC 中断应用举例

LPC2138 通过向量中断控制器（VIC）管理中断。外设中断信号需要经过两次使能控制才能到达 ARM 内核并产生异常，第一层是只有在 VIC 中使能了相应外设的中断，才能到达 VIC 并向内核发送中断请求；第二层是只有使能了内核 IRQ 中断或者 FIQ 中断，内核才能真正产生中断异常。

向量中断控制器 VIC 将所有向量 IRQ 和非向量 IRQ“相或”后向 ARM 内核提出 IRQ 请求，产生中断，程序跳转到异常向量表 0x00000018 处，读取寄存器 VICVectAddr 的内容，并以此作为地址值，跳转到中断服务程序进行中断处理。

对于向量中断，寄存器 VICVectAddr 中的内容是最高优先级 IRQ 服务程序地址（来自于寄存器 VICVectAddr0～VICVectAddr15 中的一个），对于非向量中断，寄存器 VICVectAddr 中的内容是所有非向量 IRQ 共用的默认程序地址（来自于寄存器 VICDefVectAddr）。

3.7.1 外部中断

CPU 进入掉电状态后，会自动关闭并断开 PLL。CPU 从掉电状态唤醒后，PLL 不会自动连接，需要用户重新连接。在用户对 PLL 进行重设定之前，CPU 将直接使用晶体振荡器输入频率 Fosc 作为时钟系统。

使能寄存器 INTWAKE 中的相应位，可以允许外部中断唤醒掉电的 CPU，这时可以产生中断，也可以不产生中断。如果仅仅是唤醒掉电的 CPU，则只需将 EINTn 映射到相应的引脚即可，不需要在 VIC 控制器中使能外部中断，如果唤醒后的系统不需要 IRQ 中断，甚至连 IRQ 中断都不需要打开。

例 3-4 采用外部中断唤醒掉电的 CPU。

先在启动代码中设定 PLL，使 CPU 时钟为外接晶体频率的 5 倍。主程序将按键 KEY1 连接到外部中断 0，并设置外部中断 0，唤醒 CPU。掉电前，LED1 以较快速度闪烁 5 次，然后进入掉电模式。第一次按下按键 KEY1 直接唤醒 CPU，不产生中断，也不进行 PLL 重设定，让 LED1 闪烁 5 次，可以看到 LED1 闪烁速度比掉电前慢了许多。再次进入掉电模式，第二次按下按键 KEY1，唤醒 CPU，同时产生非向量中断，在外部中断 0 服务程序中，重新设定 PLL，恢复 CPU 时钟为外接晶体频率的 5 倍，让 LED1 闪烁 5 次，可以看到，此时 LED1 闪烁速度将变得与以前一样快。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include   <intrinsics.h>

#define LED1 1 << 18                              // P0.18 控制 LED1

/****************************************************************************************
** 函数名称：IRQ_Eint0()
** 函数功能：外部中断 0 服务程序
****************************************************************************************/
void IRQ_Eint0 (void)
{ /* 等待外部中断信号恢复为高电平,若信号保持为低电平，中断标志将一直置位。 */
      while ((EXTINT & 0x01) != 0)
      {     EXTINT = 0x01;                        // 清除 EINT0 中断标志
      }
      PLLCON = 1;                                 // 使能 PLL
      PLLCFG = 0x24;                              // 设置 M 为 5，P 为 2，和掉电前一样
      PLLFEED = 0xaa;                             // 发送 PLL 馈送序列
      PLLFEED = 0x55;
      while((PLLSTAT & (1 << 8)) == 0);           // 等待 PLL 锁定
      PLLCON = 3;                                 // PLL 使能和连接
      PLLFEED = 0xaa;                             // 发送 PLL 馈送序列
      PLLFEED = 0x55;
      VICVectAddr = 0;                            // 向量中断结束
}

/****************************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时
** 入口参数：dly  延时参数，值越大，延时越久
****************************************************************************************/
void DelayNS (uint32 dly)
{     uint32 i;
      for ( ; dly>0; dly--)
      for (i=0; i<50000; i++);
}

/****************************************************************************************
** 函数名称：main()
** 函数功能：掉电唤醒后重新连接 PLL
***************************************************************************************/
int main (void)
{     uint32 i;
      TargetResetInit();                          // 在启动代码 Target.C 中设置 PLL
      PINSEL1 = 0x00000001;                       // P0.16 连接 EINT0
      IO0DIR = LED1;                              // 设置 LED1 控制口为输出，其余输入
```

```
    INTWAKE = 0x01;                 // 外部中断 0 唤醒芯片
    EXTINT = 0x01;                  // 清除 EINT0 中断标志

    for (i=0; i<5; i++)             // 掉电前 LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
    {   IO0SET = LED1;
        DelayNS(5);
        IO0CLR = LED1;
        DelayNS(5);
    }
    PCONP = 0x00;                   // 关闭定时器、RTC、UART 等功能部件
    PCON = 0x02;                    // 系统进入掉电模式，PLL 被关闭

/* 第一次按 KEY1 唤醒 CPU，不产生中断，不重设 PLL */
    for (i=0; i<5; i++)             // 唤醒后 LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
    {   IO0SET = LED1;
        DelayNS(5);
        IO0CLR = LED1;
        DelayNS(5);
    }
    PCONP = 0x00;                   // 关闭定时器、RTC、UART 等功能部件
    PCON = 0x02;                    // 系统再次进入掉电模式

/* 第二次按 KEY1 唤醒 CPU，产生非向量中断，在外部中断 0 服务程序中重设 PLL */
    __enable_irq();                 // 打开 EINT0 中断(使用非向量 IRQ)
    VICDefVectAddr = (uint32)IRQ_Eint0; // 设置非向量中断服务程序地址
    EXTINT = 0x01;                  // 清除 EINT0 中断标志
    VICIntEnable = 1 << 0x0e;       // 使能 EINT0 中断

    for (i=0; i<5; i++)             // 唤醒后 LED1 闪烁 5 次，注意观察 LED1 的闪烁频率
    {   IO0SET = LED1;
        DelayNS(5);
        IO0CLR = LED1;
        DelayNS(5);
    }
    while (1);
}
```

3.7.2 向量中断

向量 IRQ 中断处理过程如下：

- 程序正常运行。
- 发生中断。
- 跳转到异常向量表 0x00000018 处。
- 读取寄存器 VICVectAddr 的值（来自于 VICVectAddr0～VICVectAddr15 中的一个）作为中断服务程序地址。

● 跳转到中断服务处理程序，处理中断。

● 返回原中断点。

例 3-5 向量中断。

设置 PINSEL1 让 P0.16 连接外部中断 0，并设置 EXTMODE 为低电平触发，然后进行中断初始化，设置 VICIntSelect 连接外部 IRQ 中断，设置 VICVectCntl0，连接外部中断到通道 0，把中断服务程序地址放入 VICVectAddr0，即使用向量中断 0，最后在 VICIntEnable 中使能外部中断 0。全速运行程序，每按一下 KEY1 键，LED1 的状态改变一次。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>

#define LED1 1 << 18                    // P0.18 控制 LED1

/*****************************************************************************
** 函数名称：IRQ_Eint0()
** 函数功能：外部中断 0 服务程序
*****************************************************************************/
void IRQ_Eint0 (void)
{     uint32 i;
      i = IO0SET;                        // 读取当前 LED1 的控制值
      if ((i & LED1) == 0) IO0SET = LED1;
      else IO0CLR = LED1;

      while ((EXTINT & 0x01) != 0)      // 等待外部中断信号恢复为高电平
      {     EXTINT = 0x01;              // 清除 EINT0 中断标志
      }
      VICVectAddr = 0;                  // 向量中断结束
}

/*****************************************************************************
** 函数名称：main()
** 函数功能：掉电唤醒后重新连接 PLL
*****************************************************************************/
int main (void)
{     PINSEL1 = 0x00000001;                 //  P0.16 连接 EINT0
      IO0DIR = LED1;                        // 设置 LED1 控制口为输出，其余输入
      EXTMODE = 0x00;                       // 设置 EINT0 为电平触发，极性寄存器使用默认值 0
      IO0SET = LED1;
      __enable_irq();                       // 开中断
/* 打开 EINT0 中断(使用向量中断) */
      VICIntSelect = 0x00000000;            // 设置所有中断分配为 IRQ 中断
      VICVectCntl0 = 0x20 | 0x0e;           // 分配外部中断 0 到向量中断 0
      VICVectAddr0 = (uint32)IRQ_Eint0;     // 设置中断服务程序地址
```

```
        EXTINT = 0x01;                          // 清除 EINT0 中断标志
        VICIntEnable = 1 << 0x0e;               // 使能 EINT0 中断
        while (1);                              // 等待中断
    }
```

3.7.3 嵌套中断

ARM 处理器中断体系结构比较特殊，一般情况下高优先级中断并不能中断低优先级的服务程序，当多个中断同时发生的时候，只处理高优先级中断而忽略低优先级中断。

ARM 处理器的中断不会实现自动嵌套，而是通过修改启动代码中的 IRQ.S 文件来实现，用户可以根据需要，在 IRQ.S 文件末尾添加处理嵌套中断的句柄。实现嵌套中断的原则是：低优先级中断嵌套高优先级中断。

嵌套中断服务程序编写方法如下：

- 保存当前中断使能寄存器 VICIntEnable 的值。
- 禁止当前中断和低优先级中断。
- 清除中断逻辑，使 VIC 能够响应更高优先级的中断。
- 执行中断处理服务程序。
- 恢复保存的中断使能寄存器的值。

声明嵌套中断服务程序时，不能使用__irq 关键字，而是通过句柄方式实现，即在文件 IRQ.S 末尾添加如下句柄：

```
Timer0_Handler    HANDLER    IRQ_Timer0
Eint0_Handler     HANDLER    IRQ_Eint0
```

例 3-6 嵌套中断。

开启两个中断，定时器 0 中断和外部中断 0，分配定时器 0 中断优先级高于外部中断 0 的优先级。在定时器 0 中断服务程序中取反 LED1，在外部中断 0 服务程序中取反 LED2。设置 KEY1 连接外部中断 0，按住 KEY1 键后全速运行程序，看 LED1 能否闪烁，若闪烁则说明中断嵌套成功。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>

#define LED1 1 << 18  // P0.18  控制 LED1
#define LED2 1 << 19  // P0.19  控制 LED2

void Timer0_Handler(void);
void Eint0_Handler(void);
/*******************************************************************************
**  函数名称：IRQ_Timer0()
**  函数功能：Timer0 中断服务函数，取反 LED1 控制口
*******************************************************************************/
```

```
void IRQ_Timer0(void)
{       uint32 i, bak;
        bak = VICIntEnable;                       // 备份当前 VICIntEnable 的值
        VICIntEnClr = (1 << 4)|( 1<< 14);         // 禁止当前优先级中断及低优先级中断
        VICVectAddr = 0x00;             // 清除中断逻辑，以便 VIC 可以响应更高优先级 IRQ 中断
        i = IO0SET;                               // 读取当前 LED1 控制值
        if( (i & LED1) == 0 ) IO0SET = LED1;  // 控制 LED1 输出
        else IO0CLR = LED1;
        while( (T0IR & 0x01) !=0 )                // 等待外部中断信号恢复为高电平
        {       T0IR = 0x01;                      // 清除 EINT0 中断标志
        }
        VICIntEnable = bak;
}

/*****************************************************************************************
** 函数名称：IRQ_Eint0()
** 函数功能：外部中断 EINT0 服务函数，取反 LED2，允许中断嵌套
*****************************************************************************************/
void IRQ_Eint0(void)
{       uint32 i, bak;
        bak = VICIntEnable;
        VICIntEnClr = 1 << 14;                        // 禁止当前中断
        VICVectAddr = 0x00;
        IRQEnable();                                  // 使能 IRQ 中断
        i = IO0SET;                                   // 读取当前 LED2 控制值
        if( (i & LED2) == 0) IO0SET = LED2;           // 控制 LED2 输出
        else IO0CLR = LED2;
        while( (EXTINT & 0x01) != 0 )                 // 等待外部中断信号恢复为高电平
        {       EXTINT = 0x01;                        // 清除 EINT0 中断标志
        }
        VICIntEnable = bak;
}
/*****************************************************************************************
** 函数名称：main()
** 函数功能：初始化定时器 0，设置定时器 0 中断和外部中断 0
*****************************************************************************************/
int main(void)
{       T0TC = 0;                       // 定时器设置为 0
        T0PR = 0;                       // 时钟不分频
        T0MCR = 0x03;                   // 设置 T0MR0 匹配后复位 T0TC，并产生中断标志
        T0MR0 = Fpclk / 10;             // 0.1s 定时
        T0TCR = 0x01;                   // 启动定时器
        PINSEL0 = 0x00000000;
        PINSEL1 = 0x00000001;           // 引脚 P0.16 设置为 EINT0
        IO0DIR = (LED1 | LED2);         // 设置 P0 口为输出
        __enable_irq();                 // 使能 IRQ 中断
```

```
/*  打开 Timer0、EINT0 中断(设置向量控制器，使用向量 IRQ) */
      VICIntSelect = 0x00000000;               // 所有中断分配为 IRQ 中断
      VICVectCntl0 = 0x20 | 4;                 // 分配 Timer0 中断到向量中断 0
      VICVectAddr0 = (uint32)Timer0_Handler;   // 设置 Timer0 中断服务程序地址
      VICVectCntl1 = 0x20 | 14;                // 分配 EINT0 中断到向量中断 1
      VICVectAddr1 = (uint32)Eint0_Handler;    // 设置 EINT0 中断服务程序地址
      T0IR = 0x01;                             // 清除 Timer0
      EXTINT = 0x01;                           // 清除 EINT0 中断标志
      VICIntEnable = (1 << 4) | (1 << 14);     // 使能 Timer0、EINT0 中断
      while(1);                                // 等待中断
}
```

3.7.4 快速中断

快速中断 FIQ 处理过程如下：

- 程序正常运行。
- 发生中断。
- 跳转到异常向量表 0x0000001C 处。
- 读取 FIQ 中断服务程序的地址。
- 跳转到中断服务处理程序，处理中断。
- 返回原中断点。

例 3-7 快速中断。

FIQ 中断程序的写法和 IRQ 中断服务程序的写法不同，本例中主程序设定外部中断为 FIQ，在 FIQ 中断服务程序中更改 LED1 的状态。全速运行程序，每按一次 KEY1，LED1 的状态改变一次。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define KEY1 (1<<16)
#define LED1 (1<<18)

/*********************************************************************************
* 名称：DelayNS()
* 功能：延时
* 入口参数：dly        延时参数，值越大，延时越久
*********************************************************************************/
void   DelayNS(uint32   dly)
{
    uint32   i;
    for(; dly>0; dly--)
        for(i=0; i<50000; i++);
```

```
}

/********************************************************************************
** 函数名称: FIQ_Handler
** 功能描述: 快速中断异常处理程序，控制 LED 的亮与灭
********************************************************************************/
__irq void FIQ_Handler(void)
{
     uint32   i;
      i = IO0SET;                                 // 读取当前 LED1 的控制值
      if ((i & LED1) == 0) IO0SET = LED1;
      else IO0CLR = LED1;
     /* 等待外部中断信号恢复为高电平，若信号保持为低电平，中断标志会一直置位 */
      while ((EXTINT & 0x01) != 0)
      { EXTINT = 0x01;                           // 清除 EINT0 中断标志
      }
}

/********************************************************************************
* 名称：main()
* 功能：每按一次键，对相应的 LED 进行取反
********************************************************************************/
int   main(void)
{
    PINSEL0 = 0x00000000;                        // 设置所有引脚连接 GPIO*/
    PINSEL1 = 0x00000001;                        // 设置引脚连接，P0.16 为 EINT0
    IO0DIR   = LED1;
    IO0SET   = LED1;
     /* 打开 EINT0 中断(使用非向量 IRQ) */
    EXTMODE   = 0x00;                            // 设置 EINT3 中断为边沿触发模式
    EXTPOLAR = 0<<0;                             // 设置 EINT3 中断为下降沿触发模式
    __enable_fiq();                              // FIQ 中断使能

   /* 打开 EINT0 中断(使用 FIQ 中断) */
       VICIntSelect    = 1 << VIC_EINT0;         // 设置 EINT3 分配为 FIQ 中断
       EXTINT           = 0x01;                  // 清除 EINT3 中断标志
       VICIntEnable    = 1 << VIC_EINT0;         // 使能 EINT3 中断
       while(1);
}
```

3.7.5 软件中断

VIC 控制器有软件中断功能，允许采用软件强制产生与相应中断源相关的中断。在执行任何逻辑操作之前，VIC 将寄存器 VICSoftInt 的内容与 32 个不同外设的中断请求相“或”，然后再执行相应操作；通过寄存器 VICSoftIntClr 可以强制解除 VICSoftInt 中相应的中断请求。

例 3-8 软件中断。

设置外部中断 EINT0 对应的软件中断和硬件中断使能，先通过软件中断，强制实现 EINT0 中断服务程序控制 LED1 闪烁，然后再等待硬件中断。全速运行程序后，将看到 LED1 先闪烁 5 次，以后每按一次 KEY1，LED1 的状态改变一次。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define LED1 1 << 18                         // P0.18  控制 LED1

/*********************************************************************************
**  函数名称：DelayNS()
**  函数功能：软件延时
**  入口参数：dly  延时参数，值越大，延时越久
*********************************************************************************/
void DelayNS (uint32 dly)
{     uint32 i;
      for ( ; dly>0; dly--)
      for (i=0; i<50000; i++);
}

/*********************************************************************************
**  函数名称：IRQ_Eint0()
**  函数功能：外部中断  0 EINT0  服务程序，取反 BEEP  控制口
*********************************************************************************/
void   IRQ_Eint0 (void)
{     if ((IO0SET & LED1) == 0)              //  读取当前 LED1 的控制值
      IO0SET = LED1;
      else IO0CLR = LED1;
/*  等待外部中断信号恢复为高电平，若信号保持为低电平，中断标志将一直置位。*/
      while ((EXTINT & 0x01) != 0)
      {     EXTINT = 0x01;                   //  清除 EINT0 中断标志
      }
      VICSoftIntClr = 1 << 14;               //  清除软中断
      VICVectAddr = 0;                       //  向量中断结束
}

/*********************************************************************************
**  函数名称：main()
**  函数功能：初始化外部中断为非向量中断，低电平触发，然后等待中断
*********************************************************************************/
int main (void)
{     uint32 i;
      PINSEL1 = 0x00000001;                  //  设置 P0.16 为 EINT0
```

```
    IO0DIR = LED1;                          // 设置 LED1 控制口为输出，其余输入
    EXTMODE = 0x00;                         // 设置 EINT0 为电平触发，极性寄存器使用默认值 0
    __enable_irq();                         //  IRQ 中断使能
/*  打开 EINT0  中断(使用向量中断) */
    VICIntSelect = 0x00000000;              // 设置所有中断分配为 IRQ  中断
    VICVectCntl0 = 0x20 | 14;               // 分配外部中断 0  到向量中断 0
    VICVectAddr0 = (uint32)IRQ_Eint0;       // 设置中断服务程序地址
    EXTINT = 0x01;                          // 清除 EINT0 中断标志
    VICIntEnable = 1 << 14;                 // 使能 EINT0 中断
/*  强制产生 10 次软件中断  */
    for (i=0; i<10; i++)
    {   VICSoftInt = 1 << 14;               // 软件强制产生 EINT0 中断
        DelayNS(5);
    }
    while (1);                              // 等待硬件中断
}
```

复习思考题

1．LPC2138 处理器具有哪些主要特性？

2．画出 LPC2138 处理器的系统存储器地址空间映射图。

3．存储器映射的基本概念是什么？

4．LPC2138 处理器支持哪几种存储器映射方式？

5．LPC2138 处理器的存储器加速模块 MAM 提供哪几种操作模式？

6．LPC2138 处理器在使用片内锁相环 PLL 时，输入时钟频率将被限制到什么范围？

7．一个 LPC2138 处理器系统，要求 Fosc=10MHz，Fcclk=60MHz，试进行 PLL 配置。

8．说明如何使 LPC2138 处理器进入空闲模式和掉电模式。

9．说明如何将 LPC2138 处理器从空闲模式和掉电模式下唤醒。

10．向量中断控制器 VIC 可以将所有的中断请求分为哪几类？

11．向量中断控制器 VIC 使用中需要注意哪几点？

12．如果要将 LPC2138 处理器的 P0.0 和 P0.1 引脚配置为 TxD0 和 Rxd0 时，需要使用哪个引脚选择寄存器？

13．如果要将 LPC2138 处理器的 P0.27 引脚配置为 AD0.0 时，需要使用哪个引脚选择寄存器？

14．仿照例 3-2 采用中断方式设置锁相环。

15．仿照例 3-3 验证存储器加速模块功能。

16．仿照例 3−4 采用外部中断唤醒掉电的 CPU。

17．仿照例 3−5 设置向量中断。

18．仿照例 3−6 设置嵌套中断。

19．仿照例 3−7 设置快速中断。

20．仿照例 3−8 设置软件中断。

第 4 章　LPC2138 片内集成功能应用技术

ARM 核处理器 LPC2138 芯片具有丰富的片内集成功能，譬如 GPIO 接口、定时器、脉宽调制模块、A-D 转换器、D-A 转换器、UART、I^2C、SPI 和 SSP、实时时钟、看门狗定时器、系统控制模块等。每一种片内集成功能都具有相关控制寄存器，用户只要根据自己的需要，对相关寄存器进行必要的配置，就可以方便灵活地实现这些集成功能的应用。芯片引脚大多具有一个以上的功能，可以通过引脚连接模块连接到不同的片内功能上，大部分引脚复位后默认为 GPIO。

4.1　通用输入输出端口 GPIO

4.1.1　主要特性

LPC2138 芯片具有 2 个 32 位的通用 I/O 口，分别为 P0[31:0]和 P1[31:16]，其中，P0.24 未用，P0.31 仅为输出口。选择引脚 GPIO 功能时，先要配置引脚连接模块寄存器 PINSEL0、PINSEL1、PINSEL2（详见第 3 章 3.5 节），然后通过端口方向寄存器 IODIR 设置引脚属性，才能进行输入/输出操作。

GPIO 端口的主要特性如下：

- 单个位的方向控制。
- 单独控制输出的置位和清零。
- 所有 GPIO 端口在复位后默认为输入。

4.1.2　寄存器描述

当引脚配置为 GPIO 功能时，有 4 个寄存器用于控制 GPIO 的使用：IODIR、IOSET、IOCLR 和 IOPIN。IODIR 用于设置端口方向，IOSET 用于口线置 1；IOCLR 用于口线清零；IOPIN 则反映当前 I/O 口的状态。

1．GPIO 方向寄存器 IODIR

IODIR 寄存器用于控制引脚的方向。IODIR 寄存器的功能如表 4−1 所列，只有当引脚选择 GPIO 功能时，对 IODIR 的设置才有意义。

表 4−1　IODIR 寄存器功能

位	功 能 说 明	复 位 值
31:0	方向控制位：0＝输入，1＝输出；IO0DIR 的位 0 控制 P0.0、…、位 31 控制 P0.31	0

2．GPIO 输出置位寄存器 IOSET

IOSET 寄存器用于控制引脚输出高电平。IOSET 寄存器的功能如表 4−2 所列，写入 1 使对应引脚输出高电平，写入 0 无效。如果一个引脚被配置为输入或其他功能，写 IOSET 无效。读 IOSET 寄存器返回 GPIO 输出寄存器中的值。该值由前一次对 IOSET 和 IOCLR（或

IOPIN）的写操作决定，不反映任何外部环境对引脚的影响。

表 4-2　IOSET 寄存器功能

位	功 能 说 明	复 位 值
31:0	输出置位：IO0SET 的位 0 对应于 P0.0、…、位 31 对应于 P0.31。	0

3．GPIO 输出清零寄存器 IOCLR

IOCLR 寄存器用于控制引脚输出低电平。IOCLR 寄存器的功能如表 4-3 所列。写入 1 使对应引脚输出低电平并清零 IOSET 寄存器中相应位，写入 0 无效。如果一个引脚被配置为输入或其他功能，写 IOCLR 无效。

表 4-3　IOCLR 寄存器功能

位	功 能 说 明	复 位 值
31:0	输出清零：IO0CLR 的位 0 对应于 P0.0、…、位 31 对应于 P0.31	0

假定当前 P0.0＝1，且 IO0SET＝0x01，现在要将 P0.1 也拉高，以下语句是等价的：

```
IO0SET = 0x02;
或    IO0SET = 0x03;
或    IO0SET | = 0x02;
```

因为往 IO0SET 写入 0 无效。对于 IO0CLR 也有类似的情况。

这是一种普遍情况：当往某个寄存器只有写入特定逻辑的值才有效，而写入反逻辑的值无效时（如只有写入 1 有效而写入 0 无效），只操作其中某一位或者某些位时，可以不必关心其他位的状态。一般情况下，有这种特性的寄存器，均会有置位和清零寄存器成对出现，如 IO0SET 和 IO0CLR。

4．GPIO 引脚值寄存器 IOPIN

IOPIN 寄存器用于提供 GPIO 引脚的值，它反映了外部环境对引脚的影响，不能反映非 GPIO 配置引脚状态，因此不能使用 IOPIN 寄存器来监控非 GPIO 配置引脚。IOPIN 寄存器的功能如表 4-4 所列。

表 4-4　IOPIN 寄存器功能

位	功 能 说 明	复 位 值
31:0	GPIO 引脚值：IO0PIN 的位 0 对应于 P0.0、…、位 31 对应于 P0.31	0

4.1.3　应用举例

GPIO 应用时，有无需上拉电阻和需要上拉电阻两种情况。利用 GPIO 采用灌电流方式驱动 LED 时，无需上拉电阻，只要加一定阻值的限流电阻即可。这种应用中要将相应端口设置为输出，为了保护 IO 引脚，限流电阻的阻值稍大些为好，通常取 470Ω～1kΩ。

GPIO 用于输入时（例如检测按键），由于 I/O 端口内部没有上拉电阻，需要加 10kΩ左右的上拉电阻，把 I/O 口拉到高电平。另外，具有 I^2C 总线功能的 I/O 端口为开漏输出，在用作 I^2C 总线以及其他功能时，需要加 1～10kΩ的上拉电阻。

1．设置 P0.0 口为输出模式

下列语句设置 P0.0 为 GPIO 输出。

```
PINSEL0 = 0x00000000;              // P0.0 选择 GPIO 功能
IO0DIR = 0x00000001;               //  设置 P0.0 为输出
```

2．GPIO 读写操作

下列语句将读取 P0.7～P0.4 引脚值，然后从 P0.0～P0.3 输出。

```
bak = IO0PIN;                      //  读取引脚值
IO0CLR = 0x0000000F;               //  从 P0.0～P0.3 输出 0
IO0SET = (bak & 0x000000F0) >> 4;  //  从 P0.0～P0.3 输出 P0.4～P0.7 的值(为 1 的位输出 1)
```

GPIO 引脚的输出状态由写入 IOSET 或 IOCLR 寄存器的值决定，两者中后写入的寄存器值将决定引脚的最终输出状态。

通过写入 IOSET 和 IOCLR 寄存器可以使选定的输出引脚变为高/低电平状态，只有写入 1 的位对应的引脚才能设置为高/低电平，写入 0 的位对应的引脚状态不发生改变。仅通过写 IOSET 或 IOCLR 寄存器，不能使一个 GPIO 端口同时输出包含 0 和 1 的二进制数，即不能同时将某些引脚设为高电平而将某些引脚设为低电平。

写 IOPIN 寄存器可以在并行 GPIO 端口同时输出所需的二进制值。写入 IOPIN 寄存器的二进制数据将影响所有被配置为输出的引脚状态：写入 0 使引脚输出低电平，写入 1 使引脚输出高电平。如果只希望改变端口中几根引脚的状态，必须将 IOPIN 内容读出并与一个屏蔽值相与（该值使要改变的引脚对应位为 0，其他引脚对应位为 1），然后将相与的结果和期望得到的引脚输出对应的二进制数相或，最后将相或的结果写回到 IOPIN 寄存器中。

GPIO 的 Proteus 仿真电路如图 4-1 所示。

图 4-1　GPIO 的 Proteus 仿真电路

例 4-1　GPIO 输出实现流水灯。

将 P0 端口引脚连接至 GPIO，并设置为输出模式，然后通过 IO00CLR 和 IO0SET 控制接到 P0 端口引脚 LED 的亮灭，形成流水灯，显示花样可以通过数组人为定义，也可通过一定算法计算实现。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include   <intrinsics.h>
/*****************************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时
** 入口参数：dly 延时参数，值越大，延时越久
*****************************************************************************************/
void DelayNS (uint32 dly)
{     uint32 i;
for ( ; dly>0; dly--)
for (i=0; i<50000; i++);
}

/*****************************************************************************************
** 函数名称：main()
** 函数功能：GPIO 输出测试，花样流水灯。
*****************************************************************************************/
const uint32 LED_TBL[] = {                              // 花样流水灯数据
      0x00, 0xFF,                                       // 全部熄灭后，再全部点亮
      0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,   // 依次逐个点亮
      0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF,   // 依次逐个叠加
      0xFF, 0x7F, 0x3F, 0x1F, 0x0F, 0x07, 0x03, 0x01,   // 依次逐个递减
      0x81, 0x42, 0x24, 0x18, 0x18, 0x24, 0x42, 0x81,   // 两个靠拢后分开
      0x81, 0xC3, 0xE7, 0xFF, 0xFF, 0xE7, 0xC3, 0x81    // 从两边叠加后递减
};

int main (void)
{     uint8 i;
      PINSEL0 = 0x00000000;                             // 设置引脚连接 GPIO
      IO0DIR = 0x000000ff;                              // 设置 P0.0～P0.7 控制口为输出
      while (1)
      {
            for (i=0; i<42; i++)                        // 花样流水灯显示
            {
                  IO0SET = ~((LED_TBL[i]));
                  DelayNS(3);
                  IO0CLR = ((LED_TBL[i]));
                  DelayNS(3);
            }
```

```
        }
    }
```

例 4-2　GPIO 输入实现按键扫描输入。

用 GPIO 作为输入时，需要外接上拉电阻。程序中设置 IODIR 使口线成为输入方式，然后读取 IOPIN 的值即可。通过检测按键 KEY1～KEY8 的状态，控制对应的 LED 亮灭。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>
/*************************************************************************
* 函数名称：DelayNS()
* 函数功能：长软件延时
* 入口参数：dly  延时参数，值越大，延时越久
*************************************************************************/
void   DelayNS(uint32   dly)
{      uint32   i;
       for(; dly>0; dly--)
       for(i=0; i<50000; i++);
}
/*************************************************************************
** 函数名称：keyscan()
** 函数功能：按键扫描，每按一次键对相应的 LED 进行取反
*************************************************************************/
void keyscan(void)
{      uint32 i,j,k;
       k=IO0PIN;
       if((k&keycon)!=keycon)
       {
              for(j=0; j<50000; j++);          //延时去抖
              k=IO0PIN;
              if((k&keycon)!=keycon)
                     {      i=(k&keycon)|(~keycon);
                            i=~i;
                            switch(i)
                            {
                                   case KEY1 :if((IO0PIN&LED1)!=LED1) IO0SET=LED1;
                                                 else IO0CLR=LED1;
                                   break;
                                   case KEY2 :if((IO0PIN&LED2)!=LED2) IO0SET=LED2;
                                                 else IO0CLR=LED2;
                                   break;
                                   case KEY3 :if((IO0PIN&LED3)!=LED3) IO0SET=LED3;
                                                 else IO0CLR=LED3;
                                   break;
```

```
                        case KEY4 :if((IO0PIN&LED4)!=LED4) IO0SET=LED4;
                                   else IO0CLR=LED4;
                        break;
                        case KEY5 :if((IO0PIN&LED5)!=LED5) IO0SET=LED5;
                                   else IO0CLR=LED5;
                        break;
                        case KEY6 :if((IO0PIN&LED6)!=LED6) IO0SET=LED6;
                                   else IO0CLR=LED6;
                        break;
                        case KEY7 :if((IO0PIN&LED7)!=LED7) IO0SET=LED7;
                                   else IO0CLR=LED7;
                        break;
                        case KEY8 :if((IO0PIN&LED8)!=LED8) IO0SET=LED8;
                                   else IO0CLR=LED8;
                        break;
                    }
                }
                while((k&keycon)!=keycon) k=IO0PIN;
     }
}

/******************************************************************************
* 函数名称：main()
* 函数功能：每按一次键，对相应 LED 的状态取反
******************************************************************************/
int   main(void)
{
   TargetResetInit();
   PINSEL0 = 0x00000000;              // 设置所有引脚连接 GPIO
   PINSEL1 = 0x00000000;
   IO0DIR = 0x000000ff;               // 设置 P0.0～P0.7 为输出，其余为输入
   IO0CLR = 0xffffffff;
   while(1)
   {
     keyscan();
   }
}
```

例 4-3 IOPIN 的特殊应用。

通过写 IOPIN 寄存器，可以让某些端口引脚同时输出包含 1 和 0 的二进制数，达到通过端口引脚同时输出高、低电平的目的。本例通过 IOPIN 寄存器控制，使 8 个 LED 灯交替闪烁。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
```

```
/*****************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时
** 入口参数：dly  延时参数，值越大，延时越久
*****************************************************************************/
void DelayNS (uint32 dly)
{      uint32 i;
       for ( ; dly>0; dly--)
       for (i=0; i<50000; i++);
}

/*****************************************************************************
** 函数名称：main()
** 函数功能：使用 IOPIN 使 8 个 LED  交替闪烁。
*****************************************************************************/
int main (void)
{
PINSEL0 = 0x00000000;                       // 选择 GPIO
       IO0DIR = 0x000000ff;                 // 设置 P0.0～P0.7 控制口为输出
              while (1)
              {
                     IO0PIN = 0xAA;         // 点亮 LED1、LED3、LED5、LED7，其余熄灭
                     DelayNS(5);            // 延时
                     IO0PIN = 0x55 ;        // 点亮 LED2、LED4、LED6、LED8，其余熄灭
                     DelayNS(5);            // 延时
              }
}
```

4.2 通用异步接收发送器 UART

4.2.1 主要特性

LPC2138 具有 2 个符合 550 工业标准的通用异步接收发送器 UART0 和 UART1，两者具有很多相似之处，但外设基地址不同，UART1 还增加了一个调制解调器（Modem）接口。

通用异步接收发送器 UARTx（x 表示 0 或 1）包含以下功能模块：

接收器模块 UxRx 监视串行输入引脚 RxD 上的有效输入。移位寄存器 UxRSR 通过 RxD 接收有效的字符。当 UxRSR 接收到一个有效字符时，将该字符传送到缓存寄存器 FIFO（UxRBR）中，等待 CPU 或主机通过主机接口进行访问。

发送器模块 UxTx 接收 CPU 或主机写入的数据，并将数据缓存到保持寄存器 FIFO（UxTHR）中。移位寄存器 UxTSR 读取 UxTHR 中的数据，并通过串行输出引脚 TxD 发送。

波特率发生器模块 UxBRG 产生发送器模块 UxTx 所需要的定时。UxBRG 模块时钟源为 VPB 时钟 Fpclk。主时钟与 UxDLL 和 UxDLM 寄存器所定义的除数相除得到发送器模块使用的时钟。

中断接口包含寄存器 UxIER 和 UxIIR。中断接口接收几个由 UxTx 和 UxRx 发出的单时钟宽度的使能信号。

UxTx 和 UxRx 的状态信息保存在 UxLSR 中。

UxTx 和 UxRx 的控制信息保存在 UxLCR 中。

通用异步接收发送器 UART 主要特性如下：

- 16 字节收发 FIFO。
- 寄存器位置符合 550 工业标准。
- 接收器 FIFO 触发点可为 1、4、8 和 14 字节。
- 内置波特率发生器。
- UART1 包含标准调制解调器接口信号。
- UART1 包含使能实现软件或硬件流控制的机制。

4.2.2 寄存器描述

UART0 包含 11 个 8 位寄存器，UART1 包含 13 个 8 位寄存器，分别介绍如下（其中 x 表示 0 或 1）。

1．UART 接收缓存寄存器 UxRBR

UxRBR 是 UART Rx FIFO 的最高字节。它包含了最早接收到的字符，可通过总线接口读出。接收数据低位在先，即 LSB（bit0）为最先接收到的数据位。如果接收到的字符小于 8 位，则未使用的高位数据 MSB 以 0 填充。

如果要访问 UxRBR，UxLCR 寄存器的除数锁存访问位（DLAB）必须为 0。

UxRBR 为只读寄存器，其功能如表 4-5 所列。

表 4-5　UxRBR 寄存器功能

位	功　能	说　明	访问条件	复 位 值
7:0	接收器缓存	包含 Rx FIFO 中最早接收到的字节	DLAB=0	未定义

由于 PE、FE 和 BI 位与 RBR FIFO 顶端的字节相对应（即下次读 RBR 时读出的字节），因此，将接收的字节及其状态位成对读出的正确方法是先读 UxLSR，再读 UxRBR。

2．UART 发送保持寄存器 UxTHR

UxTHR 是 UART Tx FIFO 的最高字节，它包含了 Tx FIFO 中最新的字符，可通过总线接口写入。发送数据低位在先，即 LSB（bit0）代表最先发送的位。

如果要访问 UxTHR，UxLCR 寄存器的除数锁存访问位（DLAB）必须为 0。

UxTHR 为只写寄存器，其功能如表 4-6 所列。

表 4-6　UxTHR 寄存器功能

位	功　能	说　明	访问条件	复 位 值
7:0	发送器保持	写该寄存器使数据保存到 Tx IFO 中。当字节到达 FIFO 最底部并且发送器就绪时，该字节将被发送	DLAB=0	未定义

3．UART 除数锁存 LSB 寄存器 UxDLL

UxDLL 寄存器与 UxDLM 寄存器一起决定 UART 的波特率。

如果要访问 UxDLL，UxLCR 寄存器的除数锁存访问位（DLAB）必须为 1。

UxDLL 为读/写寄存器，其功能如表 4-7 所列。

表 4-7　UxDLL 寄存器功能

位	功　　能	说　　明	访 问 条 件	复　位　值
7:0	除数锁存 LSB	与 UxDLM 寄存器一起决定 UART 的波特率	DLAB=1	0x01

4．UART 除数锁存 MSB 寄存器 UxDLM

UxDLM 寄存器与 UxDLL 寄存器一起决定 UART 的波特率。

除数锁存是波特率发生器的一部分，它保存了用于产生波特率时钟的 VPB 时钟（Fpclk）分频值，UxDLL 和 UxDLM 寄存器一起构成一个 16 位除数，UxDLL 包含除数的低 8 位，UxDLM 包含除数的高 8 位。由于除数不允许为 0，因此值 0x0000 被看作是 0x0001。

波特率时钟必须是波特率(baud)的 16 倍，计算公式如下：

$$16\times baud = Fpclk/(UxDLM,UxDLL)$$

如果要访问 UxDLM，UxLCR 寄存器的除数锁存访问位（DLAB）必须为 1。

UxDLM 为读/写寄存器，其功能如表 4-8 所列。

表 4-8　UxDLM 寄存器功能

位	功　　能	说　　明	访 问 条 件	复　位　值
7:0	除数锁存 MSB	与 UxDLL 寄存器一起决定 UART 的波特率	DLAB=1	0x00

5．UART 中断使能寄存器 UxIER

UxIER 寄存器用于使能 4 个 UART 中断源，其功能如表 4-9 所列。

表 4-9　UxIER 寄存器功能

位	功　　能	说　　明	访 问 条 件	复　位　值
0	RBR 中断使能	0：禁止RDA中断 1：使能RDA中断 UxIER.0 使能接收数据可用中断。它还控制字符接收超时中断	DLAB=0	0
1	THRE 中断使能	0：禁止THRE中断 1：使能THRE中断 UxIER.1使能THRE中断。该中断的状态可从UxLSR.5读出		0
2	Rx线状态中断使能	0：禁止Rx线状态中断 1：使能Rx线状态中断 UxIER.2使能Rx线状态中断。该中断的状态可从 UxLSR[4:1]读出		0
3	Modem状态中断使能（仅适用于UART1）	0：禁止Modem中断 1：使能Modem中断 U1IER 使能modem中断。中断的状态可从U1MSR[3:0]读取		0
7：4	保留			NA

6．UART 中断标识寄存器 UxIIR

UxIIR 寄存器提供状态代码用于指示一个挂起中断的中断源和优先级。在访问 UxIIR 过程中，中断被冻结。如果在访问 UxIIR 时产生了中断，该中断被记录，下次访问 UxIIR 时可读出。

UxIIR 为只读寄存器，其功能如表 4-10 所列。

表 4-10　UxIIR 寄存器功能

位	功　能	说　明	复 位 值
0	中断挂起	0：至少有1个中断被挂起 1：没有挂起的中断 UxIIR 为低有效。挂起的中断可通过 UxIIR[3:1]确定	0
3：1	中断标识	011：1. 接收线状态（RLS） 010：2a. 接收数据可用（RDA） 110：2b. 字符超时指示（CTI） 001：3. THRE 中断 000：4 Modem中断（仅适用于UART1） UxIIR.3 指示对应于Rx FIFO 和Modem信号的中断。上面未列出的UxIIR[3:1]的其他组合都为保留值（100, 101, 111）	0
5：4	保留		NA
7：6	FIFO 使能	这些位等效于UxFCR0	0

UART 中断源和中断使能关系如图 4-2 所示。

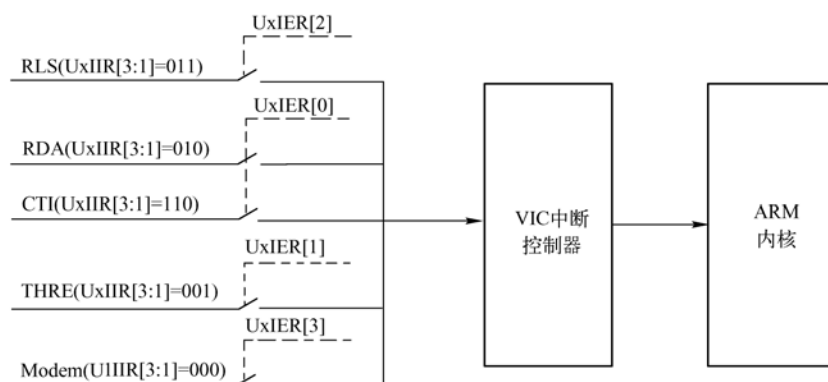


图 4-2　UART 中断源和中断使能关系

中断的处理如表 4-11 所列。只要给定了 UxIIR[3:0]的状态，中断处理程序就能确定中断源以及如何清除激活的中断。在退出中断服务程序之前，必须读取 UxIIR 来清除中断。

RLS 中断（UxIIR[3:1]=011）是最高优先级的中断。只要 Rx 输入产生 4 个错误条件（溢出错误（OE）、奇偶错误（PE）、帧错误（FE）和间隔中断（BI））中的任意一个，该中断标志将置位。产生该中断的 Rx 错误条件可通过查看 UxLSR[4:1]得到，当读取 UxLSR 时清除中断。

RDA 中断（UxIIR[3:1]=010）与 CTI 中断（UxIIR[3:1]=110）共用第二优先级。当 Rx FIFO 到达 UxFCR[7:6]所定义的触发点时，RDA 被激活；当 Rx FIFO 的深度低于触发点时，RDA 复位；当 RDA 中断激活时，CPU 可读出由触发点所定义的数据块。

CTI 中断（UxIIR[3:1]=110）为第二优先级中断。当 Rx FIFO 包含至少 1 个字符并且在接收 3.5 到 4.5 个字符的时间内没有发生 Rx FIFO 动作时，产生该中断。Rx FIFO 的任何动作（读或写 RSR）都将清除该中断。当接收到的信息不是触发值的倍数时，CTI 中断将会清空 RBR。例如，如果一个外设想要发送一个 105 个字符的信息，而触发值为 10 个字符，那么前 100 个字符将使 CPU 接收 10 个 RDA 中断，而剩下的 5 个字符使 CPU 接收 1～5 个 CTI 中断（取决于服务程序）。

THRE 中断(UxIIR[3:1[=001]为第三优先级中断。当 THR FIFO 为空并且满足特定的初始化条件时，该中断被激活。这些初始化条件将使 THR FIFO 被数据填充，以免在系统启动时产生许多 THRE 中断。初始化条件在 THRE=1 时，实现了一个字符的延时减去停止位，并在上一次 THRE=1 事件之后，没有在 UxTHR 中存在至少 2 个字符。在没有译码和服务 THRE 中断时，该延迟为 CPU 提供了将数据写入 UxTHR 的时间。如果 THR FIFO 中曾经有两个或更多字符，而当前 UxTHR 为空时，THRE 中断立即设置。当发生 UxTHR 写操作或 UxIIR 读操作，并且 THRE 为最高优先级中断（UxIIR[3:1]=001）时，THRE 中断复位。

Modem 中断（U1IIR）[3:1]=000）仅用于 UART1，它是最低优先级中断。只要在 Modem 输入引脚 RI、DCD、DSR 或 CTS 上发生任何状态变化，该中断就会被激活。此外，Modem 输入口线 RI 上低到高电平的跳变也会产生一个 Modem 中断。Modem 中断源可通过检查 U1MSR[3:0]得到。读取 U1MSR 将清除 Modem 中断。

表 4-11　中断的处理

UxIIR[3:0]	优先级	中断类型	中断源	中断复位
0001	--	无	无	--
0110	最高	Rx 线状态/错误	OE、PE、FE 或 BI	UxLSR 读操作
0100	第二	Rx 数据可用	Rx 数据可用或 FIFO 模式下（UxFCR0=1）到达触发点	UxRBR 读操作或 FIFO 低于触发值
1100	第二	字符超时指示	Rx FIFO 包含至少 1 个字符并且在一段时间内无字符输入或移出，该时间的长短取决于 FIFO 中的字符数以及在（3.5～4.5 个字符的时间内）的触发值	UxRBR 读操作
0010	第三	THRE	THRE	UxIIR 读（如果是中断源）或 THR 写操作
0000	最低	Modem 状态	RI、DCD、DSR、CTS	MSR 读操作
UxIIR[3:0]的 0011、0101、0111、1000、1001、1010、1011、1101、1110、1111 为保留值				

7．UART FIFO 控制寄存器 UxFCR

UxFCR 寄存器控制 UART Rx 和 UART Tx FIFO 的操作，其功能如表 4-12 所列。

表 4-12　UxFCR 寄存器功能

位	功　能	说　明	复 位 值
0	FIFO 使能	高电平使能对 Rx 和 Tx FIFO 及 UxFCR[7:1]的访问。该位必须置位以实现正确的 UART 操作。该位的任何变化都将使 FIFO 清空	0
1	Rx FIFO 复位	该位置位会清零 Rx FIFO 中的所有字节并复位指针逻辑。该位自动清零	0
2	Tx FIFO 复位	该位置位会清零 Tx FIFO 中的所有字节并复位指针逻辑。该位自动清零	0
5：3	保留		NA
7：6	Rx 触发选择	00：触发点0（1个字符或0x01h） 01：触发点1（4个字符或0x04h） 10：触发点2（8个字符或0x08h） 11：触发点3（14个字符或0x0Eh） 这两个位决定在激活中断之前，FIFO 中必须写入多少个字符	0

8．UART 线控制寄存器 UxLCR

UxLCR 寄存器决定发送和接收数据字符的格式，其功能如表 4-13 所列。

表 4-13　UxLCR 寄存器功能

位	功　　能	说　　明	复　位　值
1:0	字长度选择	00：5位字符长度 01：6位字符长度 10：7位字符长度 11：8 位字符长度	0
2	停止位选择	0：1个停止位 1：2个停止位（如果UxLCR[1:0]=00 则为1.5）	0
3	奇偶使能	0：禁止奇偶产生和校验 1：使能奇偶产生和校验	0
5:4	奇偶选择	00：奇数 01：偶数 10：强制为1 11：强制为0	0
6	间隔控制	0：禁止间隔发送 1：使能间隔发送 当UxLCR6=1时，输出引脚TxD强制为逻辑0	0
7	除数锁存访问位	0：禁止访问除数锁存 1：使能访问除数锁存	0

9．UART 线状态寄存器 UxLSR

UxLSR 为只读寄存器，它提供 Tx 和 Rx 模块的状态信息，其功能如表 4-14 所列。

表 4-14　UxLSR 寄存器功能

位	功　　能	说　　明	复　位　值
0	接收数据 就绪（RDR）	0：UxRBR为空 1：UxRBR包含有效数据 当 UxRBR包含未读取的字符时，UxLSR0置位；当RBR FIFO为空时，该状态位清零	0
1	溢出错误 （OE）	0：溢出错误状态未激活 1：溢出错误状态激活 溢出错误条件在错误发生后立即设置，UxLSR读操作清零该状态位 当RSR已经有新的字符就绪而RBR FIFO已满时，UxLSR1置位。此时RBR FIFO不会被覆盖， RSR中的字符将丢失	0
2	奇偶错误 （PE）	0：奇偶错误状态未激活 1：奇偶错误状态激活 当接收字符的奇偶位处于错误状态时，产生一个奇偶错误。UxLSR读操作清零该状态位。奇偶错误检测时间取决于UxFCR0。奇偶错误与RBR FIFO中读出的字符相关	0
3	帧错误 （FE）	0：帧错误状态未激活 1：帧错误状态激活 当接收字符的停止位为0时，产生帧错误。UxLSR读操作清零该状态位。帧错误检测时间取决于UxFCR0。帧错误与RBR FIFO中读出的字符相关。当检测到一个帧错误时，Rx将尝试与数据重新同步，并假设错误的停止位实际是一个超前的起始位。但即使没有出现帧错误，它也不能假设下一个接收到的字节是正确的	0
4	间隔中断 （BI）	0：间隔中断状态未激活 1：间隔中断状态激活 在发送整个字符（起始位、数据、奇偶位和停止位）过程中RxD0如果都保持逻辑0，则产生间隔中断。当检测到中断条件时，接收器立即进入空闲状态直到RxD0变为全1状态。UxLSR读操作清零该状态位。间隔检测的时间取决于UxFCR0。间隔中断与RBR FIFO中读出的字符相关	0
5	发送保持寄存 器空（THRE）	0：UxTHR 包含有效数据 1：UxTHR 空 当检测到 THR 空时，THRE 置位，UxTHR 写操作清零该状态位	1
6	发送器空 （TEMT）	0：UxTHR和/或UxTSR包含有效数据 1：UxTHR和UxTSR空 当UxTHR和UxTSR都为空时，TEMT置位。当UxTSR或UxTHR包含有效数据时，清零该状态位	1

（续）

位	功　　能	说　　明	复 位 值
7	Rx FIFO错误（RXFE）	0：UxRBR中没有Rx错误，或UxFCR0=0 1：UxRBR包含至少一个Rx错误 当一个带有Rx错误（例如帧错误、奇偶错误或间隔中断）的字符装入UxRBR时，UxLSR.7置位。当读取UxLSR寄存器并且FIFO中不再有错误时，清零该状态位	1

10．UART 高速缓存寄存器 UxSCR

在 UART 操作时 UxSCR 无效。用户可对该寄存器进行读/写。不提供中断接口向主机指示 UxSCR 所发生的读/写操作。UxSCR 寄存器功能如表 4-15 所列。

表 4-15　UxSCR 寄存器功能

位	功　　能	说　　明	复 位 值
7：0	—	一个可读可写的字节	0

11．UART1 Modem 控制寄存器 U1MCR

U1MCR 使能 Modem 的回写模式并控制 Modem 的输出信号，其功能如表 4-16 所列。

表 4-16　U1MCR 寄存器功能

位	功　　能	说　　明	复 位 值
0	DTR控制	选择Modem输出引脚DTR。该位在回写模式激活时读出为0	0
1	RTS 控制	选择Modem输出引脚RTS。该位在回写模式激活时读出为0	0
2	保留	保留，用户软件不要向其写入1。从保留位读出的值未被定义	NA
3	保留	保留，用户软件不要向其写入1。从保留位读出的值未被定义	NA
4	回写模式选择	0：禁止Modem回写模式 1：使能Modem回写模式 Modem回写模式提供了一个执行回写测试的诊断机制。发送器输出的串行数据在内部连接到接收器的串行输入端。输入脚RxD1对回写模式无影响，输出脚TxD1保持总为1状态。4个Modem输入（CTS、DSR、RI和DCD）与外部断开。从外部来看，Modem的输出端（RTS、DTR）无效。在内部，4个Modem输出连接到4个Modem输入。这样连接的结果是U1MSR的高4位由U1MCR的低4位驱动，而不是在正常模式下由4个Modem输入驱动。这样在回写模式下，写U1MCR的低4位就可产生Modem状态中断	0

12．UART1 Modem 状态寄存器 U1MSR

U1MSR 提供 Modem 输入信号的状态信息，U1MSR[3:0]在读取 U1MSR 时清零。需要注意的是，Modem 信号对 UART1 的操作没有直接影响，Modem 信号的操作是通过软件来实现的。

U1MSR 是一个只读寄存器，其功能如表 4-17 所列。

表 4-17　U1MSR 寄存器功能

位	功　　能	说　　明	复 位 值
0	Delta CTS	0：没有检测到Modem输入CTS上的状态变化 1：检测到Modem输入CTS上的状态变化 当输入CTS状态发生变化时，该位置位。读取U1MSR时清零	0
1	Delta DSR	0：没有检测到Modem输入DSR上的状态变化 1：检测到Modem输入DSR 上的状态变化 当输入DSR状态发生变化时，该位置位。读取U1MSR时清零	0
2	后沿RI	0：没有检测到Modem输入RI上的状态变化 1：检测到Modem输入RI上低到高的跳变 当输入RI发生低到高的跳变时，该位置位。读取U1MSR时清零	0

（续）

位	功　能	说　明	复 位 值
3	Delta DCD	0：没有检测到Modem输入DCD上的状态变化 1：检测到Modem输入DCD上的状态变化 当输入DCD状态发生变化时，该位置位。读取U1MSR时清零	0
4	CTS	清零发送状态：输入信号CTS的补码。在回写模式下，该位连接到U1MCR[1]	0
5	DSR	数据设备就绪状态：输入信号DSR的补码。在回写模式下，该位连接到U1MCR[0]	0
6	RI	响铃指示状态：输入信号RI的补码。在回写模式下，该位连接到U1MCR[2]	0
7	DCD	数据载波检测状态：输入信号DCD的补码。在回写模式下，该位连接到U1MCR[3]	0

13．UART1 发送使能寄存器 U1MSR

UART1 使能实现软件和硬件流控制，由 U1TER 控制。当 TxEn＝1 时，只要数据可用，UART1 发送器就将持续发送数据。一旦 TxEn 变为 0，UART1 发送器立刻停止工作。

U1TER 寄存器功能如表 4-18 所列。

表 4-18　U1TER 寄存器功能

位	功　能	说　明	复 位 值
6：0	保留		0
7	TxEn	该位为1（复位值）时，如果以前的数据都被发送后，写入THR的数据输出到TxD引脚。如果一个字符正在发送时该位清零，则结束这个字符的发送，后面的字符也不再发送，直到该位重新置位。换言之，该位为0将终止THR或Tx FIFO到发送移位寄存器的字符传输。当检测到硬件握手Tx允许信号出错或利用软件握手接收到一个XOFF字符（DC3）时，软件将该位清零。当检测到正确的Tx允许信号或接收到XON字符（DC1）时，软件又能将该位重新置位	0x01

4.2.3　应用举例

使用 UART 之前，必须先对引脚进行配置。进行如下设置可将 P0.0、P0.1 选择 UART0 功能：

```
PINSEL0 = 0x00000005;                          // 可能影响其他引脚连接
PINSEL0 = (PINSEL0 & (~0x0F)) | 0x05;          // 不影响其他引脚连接
```

进行如下设置可将 P0.8、P0.9 选择 UART1 功能：

```
PINSEL0 = 0x05 << 16;                                    // 可能影响其他引脚连接
PINSEL0 = ( PINSEL0 & (~(0x0F << 16)) ) | (0x05 << 16);  // 不影响其他引脚连接
```

LPC2138 的两个串口具有完全相同的寄存器，只是物理地址不一样。寄存器 UxRBR 与 UxTHR 是同一地址，但物理上是分开的，读操作时为 UxRBR，而写操作时为 UxTHR。

寄存器 UxDLL 与 UxRBR/UxTHR、UxDLM 与 UxTER 具有相同的地址。如果要访问 UxDLM、UxDLL，除数访问位 DLAB 必须为 1，若要访问 UxRBR/UxTHR、UxTER，则除数访问位 DLAB 必须为 0。

UxDLM 和 UxDLL 寄存器是波特率发生器的除数锁存寄存器，用于设置合适的串口波特率；波特率的除数计算公式如下：

$$UxDLM、UxDLL = Fpclk/(16 \times baud)$$

其中，baud 为所需要的波特率。

UxRBR 为数据接收缓冲，用于读取接收到的数据，若 FIFO 使能，串口接收到的数据会压入 FIFO 缓冲；UxTHR 为发送保存，向此寄存器写入数据时，将会引起串口数据发送，若 FIFO 使能，数据将会压入 FIFO 缓冲。

通过线控制寄存器 LCR 设置串口的工作模式，而 FCR 则用于 FIFO 的使能或者复位操作；当接收或者发送数据的时候，会产生相应的状态标志位（LSR）；通过对 IER 进行设置，可实现串口的发送、接收、出错中断等。

注意，IER 中的位 0 为接收中断使能，位 1 为发送中断使能，位 2 为线状态中断使能（通讯出错中断使能），若不使能相应的中断，对应的中断不会产生，此时可以通过 LSR 读取串口的状态判断串口操作是否完成或是否成功。

UART 的基本操作方法如下：

- 设置 I/O 引脚连接到 UARTx。
- 设置串口波特率（UxDLM、UxDLL）。
- 设置串口工作模式（UxLCR、UxFCR）。
- 发送或接收数据（UxTHR、UxRBR）。
- 检查串口状态字（UxLSR）或者等待串口中断（UxIIR）。
- 利用 UART 与 PC 通信，由于 PC 串口是 RS232 电平，连接时需要使用 RS232 转换器。

UART 的 Proteus 仿真电路如图 4-3 所示。

图 4-3　UART 的 Proteus 仿真电路

例 4-4 采用查询方式，通过 UART0 接收上位机发送的字符串，然后送回上位机显示。主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include   <intrinsics.h>
#include "LPCUart.h"

#define   UART_BPS  9600                /*  定义通信波特率  */

/****************************************************************************
**  函数名称：DelayNS()
**  函数功能：长软件延时。
**  入口参数：dly  延时参数，值越大，延时越久
****************************************************************************/
void DelayNS (uint32 dly)
{      uint32 i;
for ( ; dly>0; dly--)
      for (i=0; i<50000; i++);
}

/****************************************************************************
*  函数名称：UART0_Init()
*  功能描述: 初始化串口 0。
*          使用外部 25MHz 晶振，根据 CONFIG.H 文件配置，Fpclk=25MHz
*          通信波特率 9600，8 位数据位，1 位停止位，无奇偶校验
****************************************************************************/
void   UART0_Init(void)
{
    uint16 Fdiv;
    U0LCR = 1 << LCR_DLAB_BIT |                          // DLAB = 1，可设置波特率
              WordLength8 << LCR_WORDLENTH_BIT;          // WordLenth = 3, 8 位数据位
    Fdiv = (Fpclk / 16) / UART_BPS;                      //  设置波特率
    U0DLM = Fdiv / 256;                                  //  设置分频寄存器高字节
    U0DLL = Fdiv % 256;                                  //  设置分频寄存器低字节
    U0LCR = 0x03;
}

/****************************************************************************
*  函数名称：putchar()
*  功能描述: 将 printf 函数重新定义到串口 0 输出
****************************************************************************/
int putchar(int data)
{
        while (!(U0LSR & 0x20));
        return (U0THR = data);
```

```
}

/*******************************************************************************
** 函数名称：UART0_GetByte()
** 函数功能：从串口接收 1B 数据，使用查询方式接收。
** 出口参数：接收到的数据
*******************************************************************************/
uint8 UART0_GetByte (void)
{      uint8 rcv_dat;
       while ((U0LSR & 0x01) == 0);              // 等待接收标志置位
       rcv_dat = U0RBR;                           // 读取数据
       return (rcv_dat);
}

/*******************************************************************************
** 函数名称：UART0_GetStr()
** 函数功能：从串口接收
** 入口参数：s 指向接收数据数组的指针， n 接收的个数
*******************************************************************************/
void UART0_GetStr (uint8 *s, uint32 n)
{      for ( ; n>0; n--)
           { *s++ = UART0_GetByte(); }
}

/*************************************************************************
* 函数名称：UART0_SendByte()
* 功能描述：向串口发送字节数据，并等待发送完毕。
* 入口参数：data  要发送的数据
*************************************************************************/
void   UART0_SendByte(uint8 data)
{
    U0THR = data;                              // 发送数据
    while ((U0LSR&0x20)==0)   ;                // 等待数据发送完毕
}

/*************************************************************************
* 函数名称：UART0_SendStr()
* 功能描述：向串口发送一字符串
* 入口参数：srt 要发送字符串的指针
*************************************************************************/
void   UART0_SendStr(uint8 const *str)
{
    while(1)
    {
        if( *str == '\0' )
            break;
```

```
            UART0_SendByte(*str++);                         // 发送数据
        }
    }

    /****************************************************************************
    * 函数名称：main()
    * 功能描述: 向串口 UART0 发送字符串
    ****************************************************************************/
    int main(void)
    {      uint8 snd[2];
           TargetResetInit();
           PINSEL0 &= 0xFFFFFFF0;
           PINSEL0 |= 0x00000005;                          // 设置 I/O 连接到 UART0

           UART0_Init();
           UART0_SendStr("ARM7 LPC2138 System\r\n");      // 调用自定义函数从 UART0 发送数据
           printf("Hello World! \r\n");                    // 调用标准库函数发送数据
           while(1)
           {
                  printf("Please Input 1 char: \r\n");     // 调用标准库函数发送数据
                  UART0_GetStr(snd,1);                     // 调用自定义函数从 UART0 接收数据
                  printf("%c \r\r\n",snd[0]);              // 调用标准库函数发送数据
           }
    }
```

上述程序中使用自定义函数 UART0_GetStr()和 UART0_SendStr()实现串行数据的输入和输出，从虚拟终端每输入一个字符都会立即显示出来。另外，也可以调用标准库函数 printf()实现从串行口输出，但在调用之前，先要利用 putchar()函数将 printf 重新定义到串口 0，这可以通过如下语句实现：

```
    /****************************************************************************
    * 函数名称：putchar()
    * 功能描述: 将 printf 函数重新定义到串口 0 输出
    ****************************************************************************/
    int putchar(int data)
    {
      while (!(U0LSR & 0x20));
      return (U0THR = data);
    }
```

使用查询方式进行通信时，CPU 必须不停地查询相应的标志位，将会占用 CPU 很大一部分时间，效率较低。而采用中断方式进行，则可以避免这些问题。

编写 UART 中断服务处理子程序时需要注意的是，当中断服务处理子程序接收到 1 次中断，它仅仅能知道 UART 产生了中断，至于是什么类型的中断，还需要按以下方式查询中断标志寄存器 UxIIR，依据不同中断源类型，进行如下不同处理：

- UxIIR [3:0] = 0110，“奇偶错等错误或间断”中断，则从“线路状态寄存器”中读取状态位，分析具体错误原因，再进行处理。
- UxIIR [3:0] = 0100，“接收缓冲寄存器就绪”中断，则从“接收缓冲寄存器”中读取数据插入接收队列。
- UxIIR [3:0] = 0010，“发送保持寄存器空”中断，则从发送队列中读取数据写到“发送保持寄存器”中。
- UxIIR [3:0] = 1100，“字符超时指示”中断，进行超时处理。

中断服务处理子程序必须判断是否有尚未处理的中断。在处理完当前的中断源类型之后，不能立即退出服务，而应当继续判断 UxIIR 寄存器最低位是否为 0，如果为 0，则表示还有尚未处理的中断，应当继续根据 UxIIR [3:0]判断中断源类型，进行处理，直到 UxIIR 的最低位为 1，最后发送中断结束命令，结束中断服务处理程序，

例 4-5 采用中断方式通信，从上位机接收数据，然后发送回上位机显示。每当从虚拟终端输入 8 个字符后会立即显示出来。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>
#include "LPCUart.h"

#define   UART_BPS  9600              // 定义通信波特率

uint8 rcv_buf[8];                     // UART0 数据接收缓冲区
volatile uint8 rcv_new;               // 接收新数据标志

/*********************************************************************************
** 函数名称：IRQ_UART0()
** 函数功能：串口 0 接收中断服务程序
*********************************************************************************/
void IRQ_UART0 (void)
{     uint8 i;
      if ((U0IIR & 0x0F) == 0x04)
            rcv_new = 1;                    // 设置接收到新的数据标志
      for (i=0; i<8; i++)                   // FIFO 使能时设为 8B 长度
      { rcv_buf[i] = U0RBR;                 // 读取 FIFO 的数据，并清除中断
      }
      VICVectAddr = 0x00;                   // 中断处理结束
}

/*********************************************************************************
** 函数名称：UART0_SendByte()
** 函数功能：向串口 0 发送 1B 数据
** 入口参数：dat 要发送的数据
*********************************************************************************/
```

```
void UART0_SendByte (uint8 dat)
{       U0THR = dat;                                    // 要发送的数据
}

/**********************************************************************************
** 函数名称：UART0_SendBuf()
** 函数功能：向串口发送 8B 数据
** 入口参数：dat  要发送的数据
**********************************************************************************/
void UART0_SendBuf (void)
{       uint8 i;
        for (i=0; i<8; i++)
                UART0_SendByte(rcv_buf[i]);
        while ((U0LSR & 0x20) == 0);                    // 等待数据发送完毕
}

/*************************************************************************
* 函数名称：UART0_Init()
* 功能描述: 初始化串口 0。使用外部 25MHz 晶振，根据 CONFIG.H 文件配置，Fpclk=25MHz
*                 通信波特率 9600，8 位数据位，1 位停止位，无奇偶校验
*************************************************************************/
void    UART0_Init(void)
{
    uint16 Fdiv;
    U0LCR = 1 << LCR_DLAB_BIT |                         // DLAB = 1，可设置波特率
            WordLength8 << LCR_WORDLENTH_BIT;           // WordLenth = 3, 8 位数据位
    Fdiv = (Fpclk / 16) / UART_BPS;                     //  设置波特率
    U0DLM = Fdiv / 256;                                 //  设置分频寄存器高字节
    U0DLL = Fdiv % 256;                                 //  设置分频寄存器低字节
    U0LCR = 0x03;
}

/*************************************************************************
* 函数名称：putchar()
* 功能描述: 将 printf 函数重新定义到串口 0 输出
*************************************************************************/
int putchar(int data)
{
   while (!(U0LSR & 0x20));
   return (U0THR = data);
}

/**********************************************************************************
** 函数名称：main()
** 函数功能：从 UART0 接收 8 个字符，并送回上位机显示
**********************************************************************************/
```

```
int main (void)
{
	PINSEL0 &= 0xFFFFFFF0;
	PINSEL0 |= 0x00000005;			// 设置 I/O 连接到 UART0

	UART0_Init();				// 串口初始化
	U0FCR = 0x81;				// 使能 FIFO，并设置触发点为 8B
	U0IER = 0x01;				// 允许 RBR 中断，即接收中断
	__enable_irq();				// 使能 IRQ 中断
/* 使能 UART0 中断 */
	VICIntSelect = 0x00000000;		// 设置所有的通道为 IRQ 中断
	VICVectCntl0 = 0x20 | 0x06;		// UART0 分配到 IRQ slot0，即最高优先级
	VICVectAddr0 = (uint32)IRQ_UART0;	// 设置 UART0 向量地址
	VICIntEnable = 1 << 0x06;		// 使能 UART0 中断
	printf("Please input 8 characters \r\n");
	while (1)
	{	if (rcv_new == 1)		// 如果有新数据
		{	rcv_new =0;		// 清除标志，以接收新数据
			UART0_SendBuf();
			printf(" \r\r\n");
			printf("Please input 8 characters \r\n");
		}
	}
}
```

4.3 I²C 接口

4.3.1 主要特性

LPC2138 具有具有两个标准的硬件 I²C 接口，I²C0 和 I²C1，可配置为主机或从机，总线时钟速率可调整，最高可支持 400Kbit/s 总线速率。使用 I²C 总线时，要将相应的引脚设置连接 SCL、SDA，并且总线上要接 2 个阻值在 1～10kΩ之间的上拉电阻。

I²C 接口的主要特性如下：

- 标准的 I²C 总线接口。
- 可配置为主机、从机或主/从机。
- 可编程时钟可实现通信速率控制。
- 主机、从机之间双向数据传输。
- 多主机总线（无中央主机）。
- 同时发送的主机之间仲裁，避免了总线数据冲突。
- 串行时钟同步使器件在一条串行总线上实现不同位速率的通信。
- 串行时钟同步可作为握手机制。

I²C 总线的典型配置如图 4-4 所示。

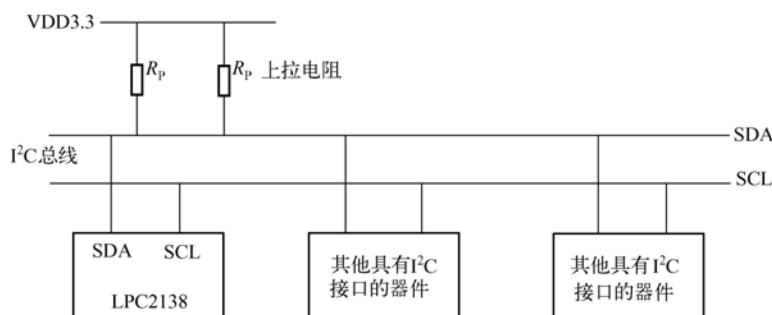


图 4-4　I^{2}C 总线的典型配置

根据方向位（R/W）状态的不同，I^{2}C 总线上存在以下两种类型的数据传输：

1）从主发送器向从接收器发送数据，即主发送。主机发送的第一个字节是从机地址，接下来是数据字节流。从机每接收一个字节返回一个应答位。

2）从发送器向主接收器发送数据，即主接收。第一个字节（从地址）由主机发送，然后从机返回一个应答位，接下来从机向主机发送数据字节。主机每接收一个字节返回一个应答位，接收完最后一个字节，主机返回一个“非应答位”。

当主机产生起始条件、发送从机寻址字节（从机地址+读写位）之后，即开始一次串行数据发送/接收。当出现停止条件时，此次传输结束。

LPC2138 是字节方式的 I^{2}C 接口，只要把一个字节数据写入 I^{2}C 数据寄存器 I2DAT 后，即可由 I^{2}C 接口自动完成所有的数据位发送。I^{2}C 接口可以配置为主机，亦可配置为从机，它有 4 种操作模式：主发送模式、主接收模式、从发送模式和从接收模式。下面分别予以介绍。

1．主发送模式

在这种模式下，数据从主机发送到从机。在进入主发送器模式之前，必须按照图 4-5 所示格式对 I2CONSET 寄存器进行初始化。其中，I2EN 位置 1 使能 I^{2}C 功能；STA、STO 和 SI 位必须为 0，分别表示起始标志为 0、停止标志为 0 和 I^{2}C 中断标志为 0；AA 位也必须为 0，不产生应答信号，即不允许进入从机模式。

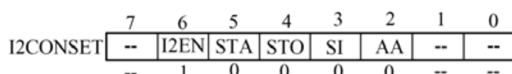


图 4-5　主模式配置

主模式 I^{2}C 的数据发送格式如图 4-6 所示。起始和停止条件用于指示串行传输的起始和结束。第一个发送的数据包含接收器件的从地址(7 位)和读写操作位。在主发送模式下，读写操作位（R/W）应当为 0，表示执行写操作。数据的发送每次为 8 位，即 1 个字节。每发送完一个字节，主机都会接收到一个由从机返回的应答位。

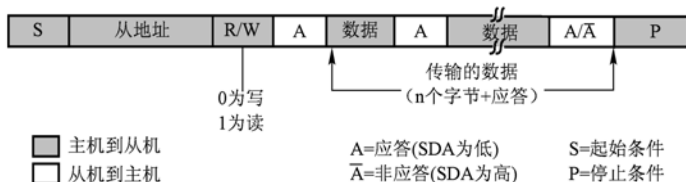


图 4-6　主发送模式下的数据格式

主发送模式下的数据发送操作步骤如下：

1）通过软件置位 STA 位，进入主发送模式，I^2C 逻辑在总线空闲后，立即发送一个起始条件。

2）发送完起始条件后，SI 位置位，此时状态寄存器 I2STAT 中的状态代码为 08H。该状态代码用于中断服务程序的处理。

3）将从地址和读/写操作位装入数据寄存器 I2DAT，清零 SI 位，开始发送从地址和 W 位。

4）当从地址和 W 位已发送且接收到应答位之后，SI 位再次置位，可能的状态代码为 18H、20H 或 38H。

5）若状态码为 18H，表示从机已应答，可以将数据装入 I2DAT 寄存器，然后清零 SI 位，开始发送数据。

6）当正确发送数据后，SI 位再次置位，可能的状态代码为 28H 或 30H，此时可以再次发送数据，或者置位 STO，结束总线操作。

2．主接收模式

在这种模式下，主机所接收的数据字节来自从机，数据接收格式如图 4-7 所示。在此模式下，读/写操作位（R/W）应该为 1，表示执行读操作。

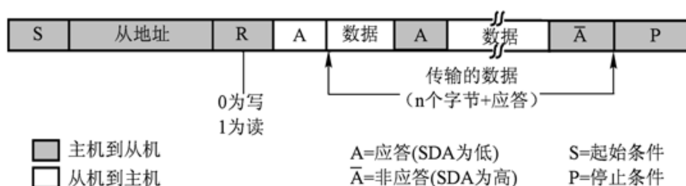


图 4-7　主接收模式下的数据格式

主模式 I^2C 的数据接收操作步骤如下：

1）通过软件置位 STA，进入主接收模式，在 I^2C 逻辑总线空闲后，立即发送一个起始条件。

2）发送完起始条件后，SI 位置位，此时状态寄存器 I2STAT 中的状态代码为 08H。该状态代码用于中断服务程序的处理。

3）将从地址和读/写操作位装入数据寄存器 I2DAT，清零 SI 位，开始发送从地址和 R 位。

4）当从地址和 R 位已发送且接收到应答位之后，SI 位再次置位，可能的状态代码为 38H、40H 或 48H。

5）若状态码为 40H，表明从机已应答。设置 AA 位，用来控制接收到数据后是产生应答信号，还是产生非应答信号，然后清零 SI 位，开始接收数据。

6）正确接收到一个字节数据后，SI 位再次置位，可能的状态代码为 50H 或 58H，此时可以再次接收数据，或者置位 STO 结束总线。

3．从接收模式

在这种模式下，数据从主机读取。为了初始化为从接收模式，用户必须将从地址写入地址寄存器 I2ADR，并按照图 4-8 所示格式对 I2CONSET 寄存器进行初始化。其中，I2EN 位置 1 使能 I^2C 功能；STA、STO 和 SI 位必须为 0，分别表示起始标志为 0、停止标志为 0 和

I^2C 中断标志为 0；AA 位为 1，应答从机地址。

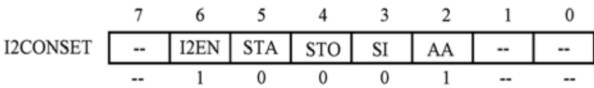


图 4-8　从模式配置

实际应用中，通常采用中断方式进行 I^2C 操作。当主机访问从机时，若读/写操作位为 0（W），则从机进入从接收模式，接收主机发送过来的数据，并产生应答信号。从接收模式下的数据接收格式如图 4-9 所示。从接收模式中，总线时钟、起始条件、从机地址、停止条件仍由主机产生。

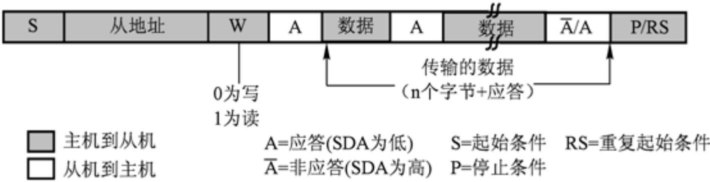


图 4-9　从接收模式下的数据格式

使用从模式 I^2C 时，用户程序只需在 I^2C 中断服务中完成各种数据操作，即根据状态码进行相应操作。当地址和读/写操作位接收后，置位 SI 位并从状态寄存器 I2STAT 中读取有效的状态码。

4．从发送模式

当主机访问从机时，若读写操作位为 1（R），则从机进入从发送模式，向主机发送数据，并等待主机的应答信号。从发送模式下的数据接收格式如图 4-10 所示。从发送模式中，总线时钟、起始条件、从机地址、停止条件仍由主机产生。

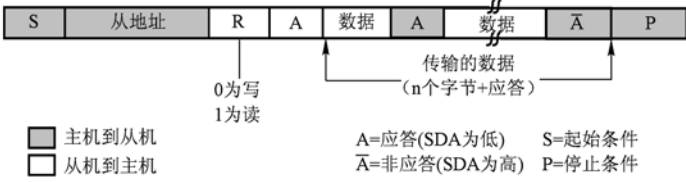


图 4-10　从发送模式下的数据格式

第一个字节的处理与从接收方式相同。但是在这种方式下，读/写操作位将标明数据传输方向相反。在具体应用中，I^2C 可能作为主机，也可能作为从机。在从模式下，I^2C 硬件查找从机地址和通用调用地址。如果检测到其中一个地址，即产生中断请求。当 CPU 变为主机时，在进入主模式之前，硬件等待总线空闲可能导致从机没有中断，如果在主模式下总线仲裁丢失，I^2C 将立即转换到从模式，不能检测到自己的从机地址。

4.3.2　寄存器描述

I^2C 接口包含 7 个寄存器，分别介绍如下（其中 x 表示 0 或 1）：

1．I^2C 控制置位寄存器 I2CxCONSET

I2CxCONSET 寄存器的功能如表 4-19 所列。

表 4-19　I2CxCONSET 寄存器功能

位	功　能	说　明	复 位 值
0	保留	用户软件不要向其写入 1，读出值未定义	NA
1	保留	用户软件不要向其写入 1，读出值未定义	NA
2	AA	应答标志	0
3	SI	I^2C 中断标志	0
4	STO	停止标志	0
5	STA	起始标志	0
6	I2EN	I^2C 接口使能	0
7	保留	用户软件不要向其写入 1，读出值未定义	NA

AA 为应答标志。当该位置位时，满足下列条件之一，将产生一个应答（SDA 上的低电平）：

- 接收到从地址寄存器中的地址。
- 当 I2CxADR 寄存器中的通用调用位（GC）置位时，接收到通用调用地址。
- 当 I^2C 接口处于主接收模式时，接收到一个数据字节。
- 当 I^2C 接口处于可寻址的从接收模式时，接收到一个数据字节。

向 I2CxCONCLR 寄存器中的 AAC 位写入 1 将使 AA 位清零。当 AA 为零时，下列情况之一将返回一个非应答信号（SDA 上的高电平）：

- 当 I^2C 接口处于主接收模式时，接收到一个数据字节。
- 当 I^2C 接口处于可寻址的从接收模式时，接收到一个数据字节。

SI 为 I^2C 中断标志。当进入 25 种可能的 I^2C 状态中的任何一种后，该位置位。通常，I^2C 中断只在空闲的从器件中用于指示一个起始条件，或在一个空闲的主器件（如果它等待使用 I^2C 总线）中指示一个停止条件。向 I2CxCONCLR 寄存器中的 SIC 位写入 1 将使 SI 位清零。

STO 为停止标志。当 STO 为 1 时，在主模式中，向 I^2C 总线发送一个停止条件或在从模式中使总线从错误状态中恢复，当主模式中 STO=1 时，向总线发送停止条件。当总线检测到停止条件时，STO 自动清零。

在从模式中，置位 STO 位可从错误状态中恢复。这种情况下不向总线发送停止条件。硬件的表现就好像是接收到一个停止条件并切换到不可寻址的从接收器模式。STO 标志由硬件自动清零。

STA 为起始标志。当 STA=1 时，I^2C 接口进入主模式并发送一个起始条件，如果已经处于主模式，则发送一个重复起始条件。

当 STA=1 并且 I^2C 接口还没进入主模式时，I^2C 接口将进入主模式，检测总线并在总线空闲时产生一个起始条件。如果总线忙，则等待一个停止条件（释放总线），并在延迟半个内部时钟周期后，发送一个起始条件。当 I^2C 接口已经处于主模式并发送或接收了数据时，I^2C 接口会发送一个重复的起始条件。STA 可在任何时候置位，当 I^2C 接口处于可寻址的从模式时，STA 也可以置位。

向 I2CxCONCLR 寄存器中的 STAC 位写入 1 将使 STA 位清零。当 STA=0 时，不会产生起始或重复起始条件。

当 STA 和 STO 都置位时，如果 I^2C 接口处于主模式，I^2C 接口将向总线发送一个停止条件，然后发送一个起始条件。如果 I^2C 接口处于从模式，则产生一个内部停止条件，但不发送到总线上。

I2EN 为 I^2C 接口使能。该位置位时，使能 I^2C 接口。向 I2CxCONCLR 寄存器中的 I2ENC 位写入 1 将使 I2EN 位清零。当 I2EN 位为 0 时，I^2C 功能被禁止。

2．I^2C 控制清零寄存器 I2CxCONCLR

I2CxCONCLR 寄存器的功能如表 4−20 所列。

表 4−20　I2CxCONCLR 寄存器功能

位	功　能	说　明	复 位 值
0	保留	用户软件不要向其写入 1，读出值未定义	NA
1	保留	用户软件不要向其写入 1，读出值未定义	NA
2	AAC	应答标志清零位，写入1清零I2CxCONSET寄存器中的AA位；写入 0 无效	0
3	SIC	I^2C中断标志清零位，写入1清零I2CxCONSET寄存器中的SI位；写入 0 无效	0
4	保留	用户软件不要向其写入 1，读出值未定义	NA
5	STAC	起始标志清零位，写入1清零I2CxCONSET寄存器中的STA位；写入 0 无效	0
6	I2ENC	I^2C接口禁止，写入1清零I2CxCONSET寄存器中的I2EN位；写入 0 无效	0
7	保留	用户软件不要向其写入 1，读出值未定义	NA

3．I^2C 状态寄存器 I2CxSTAT

I2CxCSTAT 是只读寄存器，它包含 I^2C 接口的状态代码，最低 3 位总是为 0。一共有 26 种可能存在的状态代码。当代码为 F8H 时，无可用的相关信息，SI 位不会置位。所有其他 25 种状态代码都对应一个已定义的 I^2C 状态，进入其中一种状态时，SI 位将置位。

I2CxCSTAT 寄存器的功能如表 4−21 所列。

表 4−21　I2CxSTAT 寄存器功能

位	功　能	说　明	复 位 值
2：0	状态	这 3 个位总是为 0	0
1	状态	状态位	1

4．I^2C 数据寄存器 I2CxDAT

I2CxDAT 寄存器包含要发送或刚接收的数据。当它没有处理字节的移位时，CPU 可对其进行读/写。该寄存器只能在 SI 置位时访问，在 SI 置位期间，I2CxDAT 中的数据保持稳定。I2CxDAT 中的数据移位总是从右至左进行：第一个发送的位是 MSB（位 7），在接收字节时，第一个接收到的位存放在 I2CxDAT 的 MSB。

I2CxDAT 寄存器的功能如表 4−22 所列。

表 4-22 I2CxDAT 寄存器功能

位	功 能	说 明	复 位 值
7：0	数据	发送/接收的数据	0

5．I^2C 从地址寄存器 I2CxADR

I2CxADR 寄存器可读/写，但只在 I^2C 设置为从模式时才能使用，在主模式下该寄存器无效。I2CxADR 寄存器的 LSB 为通用调用位，当该位置位时，通用调用地址（00H）被识别。

I2CxADR 寄存器的功能如表 4-23 所列。

表 4-23 I2CxADR 寄存器功能

位	功 能	说 明	复 位 值
0	GC	通用调用	0
7：1	地址	从模式地址	0

6．I^2C SCL 占空比寄存器 I2CxSCLH 和 I2CxSCLL

通过设置 I2CxSCLH 和 I2CxSCLL 寄存器可以得到不同的波特率，这两个寄存器的功能分别如表 4-24 和表 4-25 所列。

表 4-24 I2CxSCLH 寄存器功能

位	功 能	说 明	复 位 值
15：0	计数值	SCL 高电平周期选择计数	0x0004

表 4-25 I2CxSCLL 寄存器功能

位	功 能	说 明	复 位 值
15：0	计数值	SCL 低电平周期选择计数	0x0004

软件必须通过对 I2CxSCLH 和 I2CxSCLL 寄存器进行设置来选择合适的波特率。I2CxSCLH 定义 SCL 高电平所保持的 Fpclk 周期数，I2CxSCLL 定义 SCL 低电平的 Fpclk 周期数。I^2C 接口的位频率由下面公式计算得出：

位频率=Fpclk/（I2CxSCLH+I2CxSCLL）

I2CxSCLL 和 I2CxSCLH 的值不一定要相同。通过设定这两个寄存器，可得到 SCL 的不同占空比。但寄存器的值必须确保 I^2C 数据通信速率在 0～400kHz 之间。这样对 I2CxSCL 和 I2CxSCLH 的值就有一些限制，每个寄存器的值都必须大于等于 4。

4.3.3 应用举例

使用 I^2C 接口之前，首先要对引脚进行配置。进行如下设置可将 P0.2、P0.2 选择 I^2C0 功能：

```
PINSEL0 = 0x00000050;                          // 可能影响其他引脚连接
PINSEL0 = (PINSEL0 & (~0xF0)) | 0x50;          // 不影响其他引脚连接
```

进行如下设置可将 P0.11、P0.14 选择 I^2C1 功能：

```
PINSEL0 = (0x03 << 22) | (0x03 << 28);                    // 可能影响其他引脚连接
PINSEL0 = (PINSEL0 & (~(0x03 << 22))) | (0x03 << 22) |
(PINSEL0 & (~(0x03 << 28))) | (0x03 << 28);              // 不影响其他引脚连接
```

I^2C 时钟由 SCL 占空比寄存器 I2CxSCLH 和 I2CxSCLL 控制，分别用于设置 SCL 的高电平和低电平时间，以便获得合适的总线时钟速率；寄存器 I2CxCONSET 用来控制 I^2C 总线的模式及总线操作，其中位 SI 为 I^2C 中断标志，所有总线操作都要依赖于这一标志，同时它又钳住总线，使总线的数据发送/接收得以同步控制；寄存器 I2CxCONCLR 则为对应的清零寄存器；I2CxSTAT 为 I^2C 状态寄存器，用于指示总线处于哪种状态，以方便控制操作；I2CxDAT 为 I^2C 总线数据寄存器，它包含要发送的数据或刚收到的数据。当系统作为从机时，从机地址寄存器 I2CxADR 有效，当总线对此地址进行访问时，将会产生 I^2C 中断。

I^2C 主机基本操作方法如下：

- 设置 I^2C 引脚连接。
- 设置 I^2C 时钟速率（I2CxSCLH、I2CxSCLL）。
- 设置为主机，并发送起始信号（I2CxCONSET 的 I2EN、STA 位为 1，AA 位为 0）。
- 发送从机地址（I2CxDAT），控制 I2CxCONSET 发送。
- 判断总线状态（I2CxSTAT），进行数据传输控制。
- 发送结束信号（I2CxCONSET）。

I^2C 从机基本操作方法如下：

- 设置 I^2C 引脚连接。
- 设置自身的从机地址（I2CxADR）。
- 使能 I^2C（I2CxCONSET 的 I2EN、AA 位为 1）。
- 判断 SI 位或者等待 I^2C 中断，等待主机操作。
- 判断总线状态 I2CxSTAT，进行数据传输控制。

初始化 I^2C0 为主模式：

```
I2C0CONCLR = 0x2C;                  // 清除 AAC、SIC 和 STAC 位
I2C0CONSET = 0x40;                  // AA=0，主 I2C 模式，I2EN=1，I2C 使能
```

初始化 I^2C1 为从模式：

```
I2C1ADR = adr & 0xFE;               // 设置从机地址 adr
I2C1CONCLR = 0x28;                  // 清除 STAC 和 SIC 位
I2C1CONSET = 0x44;                  // AA=1，从 I2C 模式，I2EN=1，I2C 使能
```

例 4-6 利用 LPC2138 的硬件，I^2C 接口对串行 EEPROM 器件 24C02 进行读/写。Proteus 仿真电路如图 4-11 所示。程序运行后，如果对 EEPROM 器件 24C02 读/写正确，LED4 将被点亮，否则 LED1 将不断闪烁。本例用到了以中断方式操作的硬件 I^2C 软件包，包括 2 个文件，I2CINT.c 和 I2CINT.h，为减少篇幅起见，这里仅列出主程序文件。

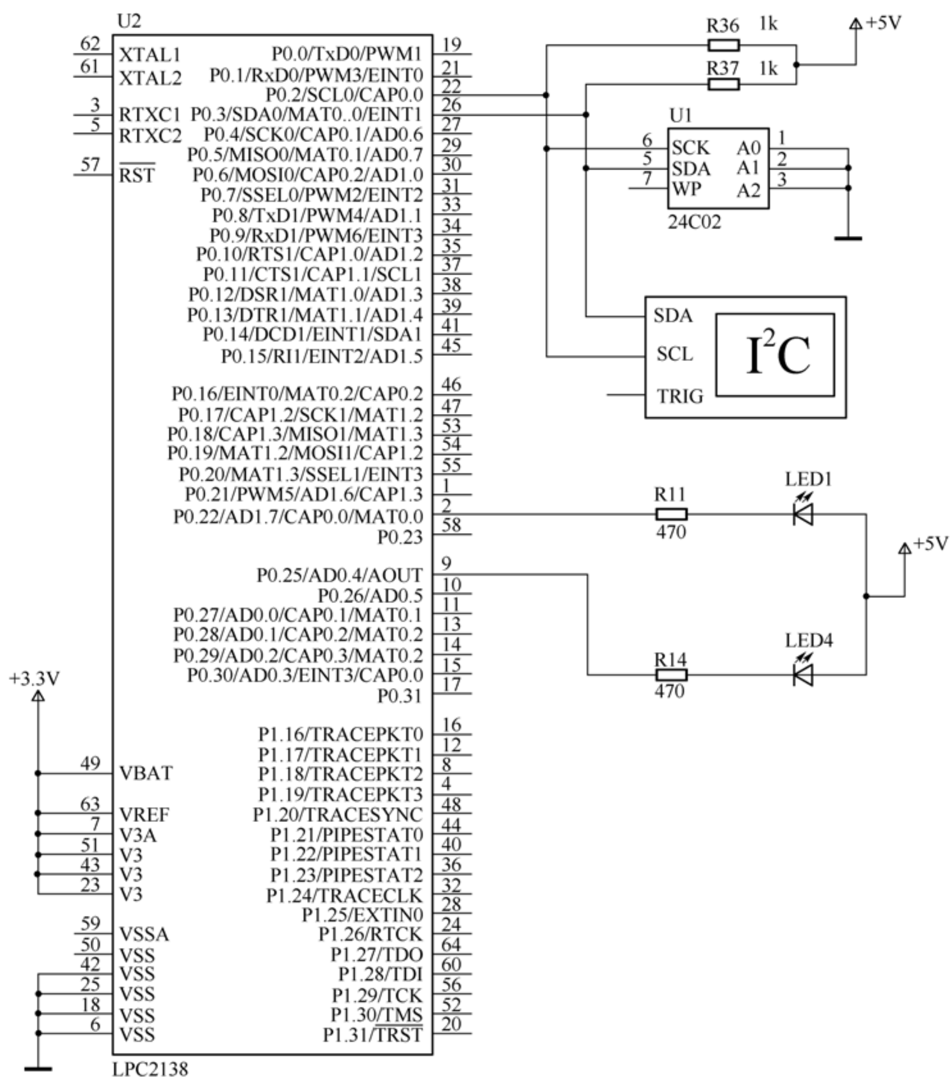


图 4-11　利用 I²C 接口读/写 24C02 的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
#include    "I2CINT.H"

#define     CSI24WC02 0xA0                      /* 定义器件地址 */
#define     LED1CON  (1<<22)                    /* P0.22 引脚控制 LED1，低电平点亮 */
#define     LED4CON  (1<<25)                    /* P0.22 引脚控制 LED4，低电平点亮 */
```

```
/******************************************************************************
* 函数名称：I2C_Init()
* 函数功能：I2C 初始化，包括向量 IRQ 中断
******************************************************************************/
void  I2C_Init(void)
{  /* 设置 I2C 时钟为 100kHz */
   I2C0SCLH = I2C0SCLL = 14;              // 晶振为 11.0592MHz，Fpclk = 2.7648MHz

   /* 设置 I2C 中断允许 */
   VICIntSelect = 0x00000000;             // 设置所有通道为 IRQ 中断
   VICVectCntl0 = 0x29;                   // I2C 通道分配到 IRQ slot 0，即优先级最高
   VICVectAddr0 = (int)IRQ_I2C;           // 设置 I2C 中断向量地址
   VICIntEnable = 0x0200;                 // 使能 I2C 中断
}

/******************************************************************************
* 函数名称：DelayNS()
* 函数功能：软件延时
* 入口参数：dly      延时参数，值越大，延时越久
******************************************************************************/
void  DelayNS(uint32  dly)
{  uint32  i;
   for(; dly>0; dly--)
      for(i=0; i<50000; i++);
}


/******************************************************************************
* 函数名称：WrEepromErr()
* 函数功能：读写 EEPRM 出错报警，闪动 LED1
******************************************************************************/
void  WrEepromErr(void)
{  while(1)
   {  IO0SET = LED1CON;
      DelayNS(10);
      IO0CLR = LED1CON;
      DelayNS(10);
   }
}

/******************************************************************************
* 函数名称：main()
* 函数功能：向 E2PROM 写入 10B 数据，然后读出，判断数据写入是否正确
******************************************************************************/
int  main(void)
```

```
{   uint8   i;
    uint8   data_buf[30];
    PINSEL0 = 0x00000050;                          // 设置 I2C 口
    PINSEL1 = 0x00000000;
    IO0DIR = LED1CON|LED4CON;                      // 设置 LED1、LED4 为输出
    IO0SET = LED1CON|LED4CON;
    __enable_irq();                                // 使能 IRQ 中断
    I2C_Init();                                    //  I2C 初始化

    for(i=0; i<10; i++) data_buf[i] = i+'0';
    ISendStr(CSI24WC02, 0x00, data_buf, 10);       // 从 0x00 地址开始写入 10B 数据
    DelayNS(1);                                    // 等待写周期结束

    for(i=0; i<10; i++) data_buf[i] = 0;
    IRcvStr(CSI24WC02, 0x00, data_buf, 10);        // 从 0x00 地址开始读出 10B 数据

    /*  校验读出的数据，若不正确则闪烁 LED1 报警  */
    for(i=0; i<10; i++)
    {   if(data_buf[i]!=(i+'0'))
            WrEepromErr();
    }

    IO0CLR = LED4CON;                              // 读出校验正确，点亮 LED4
    while(1);
}
```

4.4 SPI 接口

4.4.1 主要特性

LPC2138 具有一个硬件 SPI 接口，它是一个全双工的同步串行接口，可以配置为主机或者从机，在同一总线上可以有多个主机或者从机，但同一时刻只能有一个主机和一个从机能够进行通信，在一次数据传输过程中，主机向从机发送一字节数据，从机也向主机返回一字节数据。

SPI 接口的主要特性如下：

- 两个完全独立的 SPI 控制器。
- 遵循串行外设接口（SPI）规范。
- 同步、串行、全双工通信。
- 组合的 SPI 主机和从机。
- 最大数据位速率为输入时钟速率的 1/8。
- 可应用于串行存储器，如 DataFlash、三线 EEPROM 等。
- 可应用于串行外设，如 ADC、DAC、LCD 控制器、CAN 控制器、传感器等。

4.4.2 SPI 数据传输

图 4-12 所示为 LPC2138 硬件 SPI 接口 4 种不同数据传输格式的时序。该时序图描述的是 8 位数据传输，时序图分成了 3 个部分。第一部分描述 SCK 和 SSEL 信号；第二部分描述 CPHA=0 时的 MOSI 和 MISO 信号；第三部分描述 CPHA＝1 时的 MOSI 和 MISO 信号。

在时序图的第一部分需要注意两点：第一，时序图包含了 CPOL 置为 0 和 1 的情况；第二，SSEL 信号的激活和未激活，当 SPI 接口为从机时 SSEL 用作器件的片选信号。

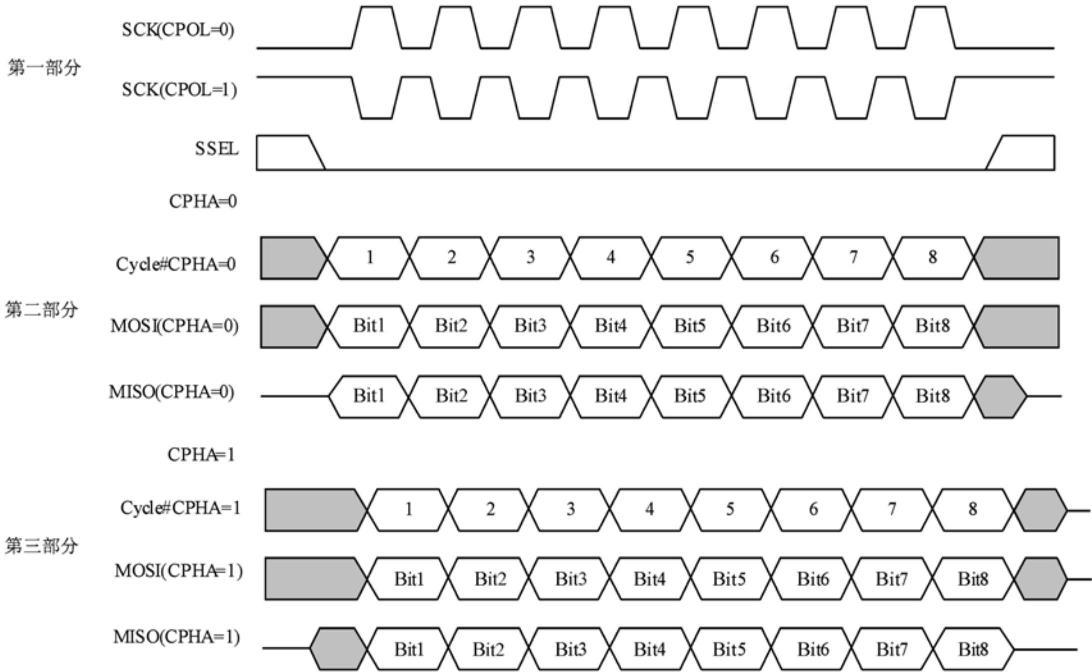


图 4-12　SPI 数据传输格式（CPHA=0 和 CPHA=1）

SPI 数据和时钟的相位关系如表 4-26 所列。该表汇集了 CPOL 和 CPHA 的每一种设定。

表 4-26　SPI 数据和时钟的相位关系

CPOL 和 CPHA 的设定	第一位数据的输出	第一位数据的输出	采样的数据
CPOL=0，CPHA=0	在第一个 SCK 上升沿之前	SCK 下降沿	SCK 上升沿
CPOL=0，CPHA=1	第一个 SCK 上升沿	SCK 上升沿	SCK 下降沿
CPOL=1，CPHA=0	在第一个 SCK 上升沿之前	SCK 上升沿	SCK 下降沿
CPOL=1，CPHA=1	第一个 SCK 下降沿	SCK 下降沿	SCK 上升沿

8 位数据传输的起始和停止，依据器件是主机还是从机以及 CPHA 变量的设置。当器件为主机时，传输的起始由准备发送 1 个字节数据的主机启动，此时，主机可激活时钟并开始传输。当传输的最后一个时钟周期结束时，传输结束。当器件为从机并且 CPHA=0 时，传输在 SSEL 信号激活时开始，并在 SSEL 变为高电平时结束；当器件为

从机且 CPHA=1 时，如果该器件被选择，传输从第一个时钟沿开始，并在数据采样的最后一个时钟沿结束。

SPI 数据传输过程中可能发生以下异常状况：

① 读溢出。当 SP 模块内部读缓冲区包含没有读出的数据，而新的传输已经完成，就会发生读溢出。SPIF 位置位表示读缓冲区包含了有效数据。当一次传输结束时，SPI 模块需要将接收到的数据移到读缓冲区。如果 SPIF 位置位（读缓冲区已满），新接收到的数据将会丢失，同时状态寄存器的读溢出（ROVR）位将置位。

② 写冲突。SPI 总线接口与内部移位寄存器之间没有写缓冲区，因而在 SPI 数据传输过程当中，从传输启动时开始，直到 SPIF 置位时读取状态寄存器为止，不能向 SPI 数据寄存器写入数据。如果在这段时间内写 SPI 数据寄存器，写入的数据将会丢失，同时状态寄存器中的写冲突位（WCOL）将置位。

③ 模式错误。SSEL 信号在 SPI 模块为主机时必须无效，不能用作 GPIO。当 SPI 模块为主机时，如果 SSEL 信号被激活（将 SSEL 变为低电平），表示有另外一个主机将该器件选择为从机，这种状态称为模式错误。当检测到一个模式错误时，状态寄存器的模式错误位（MODF）位置位，SPI 信号驱动器关闭，SPI 转换为从模式。

④ 从机中止。如果 SSEL 信号在传输结束之前变为高电平，则从传输将被认为中止。此时，正在处理的发送或接收数据都将丢失，状态寄存器的从机中止（ABRT）位将置位。

4.4.3 寄存器描述

SPI 接口包含 5 个寄存器，分别介绍如下。

1．SPI 控制寄存器 SPCR

SPCR 寄存器用于控制 SPI 模块的功能，该寄存器必须在数据传输之前进行设定，其功能如表 4−27 所列。

表 4−27 SPCR 寄存器功能

位	功　能	说　明	复 位 值
2：0	保留	用户软件不要向其写入 1，读出值未定义	NA
3	CPHA	时钟相位控制。决定 SPI 传输时数据和时钟的关系，并控制从机传输的起始和结束。为 1 时，数据在 SCK 的第二个时钟沿采样。当 SSEL 信号激活时，传输从第一个时钟沿开始并在最后一个采样时钟沿结束；为 0 时，数据在 SCK 的第一个时钟沿采样。传输从 SSEL 信号激活时开始，并在 SSEL 信号无效时结束	0
4	CPOL	时钟极性控制。为 1 时，SCK 为低有效；为 0 时，SCK 为高有效	0
5	MSTR	主模式选择。为 1 时，SPI 处于主模式；为 0 时，SPI 处于从模式	0
6	LSBF	字节移动方向。为 1 时，SPI 数据传输 LSB（位 0）在先；为 0 时，SPI 数据传输 MSB（位 7）在先	0
7	SPIE	SPI 中断使能。为 1 时，每次 SPIF 或 MODF 置位时都会产生硬件中断；为 0 时，SPI 中断被禁止	0

2．SPI 状态寄存器 SPSR

SPSR 寄存器显示 SPI 总线状态，包括一般性功能和异常状况。该寄存器的主要用途是监测数据传输是否完成（查询 SPIF 位），以及指示异常状况。该寄存器的功能如表 4−28 所列。

表 4-28　SPSR 寄存器功能

位	功　能	说　明	复 位 值
2：0	保留	用户软件不要向其写入 1，读出值未定义	NA
3	ABRT	从机中止。为 1 时表示发生了从机中止；读取该寄存器时该位清零	0
4	MODF	模式错误。为 1 时表示发生了模式错误；先通过读取该寄存器清零 MODF 位，再写 SPI 控制寄存器	0
5	ROVR	读溢出。为 1 时表示发生了读溢出；读取该寄存器时该位清零	0
6	WCOL	写冲突。为 1 时表示发生了写冲突；先通过读取该寄存器清零 WCOL 位，再访问 SP 数据寄存器	0
7	SPIF	SPI 传输完成标志。为 1 时表示一次 SPI 数据传输完成。在主模式下，该位在传输的最后一个周期置位；在从机模式下，该位在 SCK 的最后一个数据采样边沿置位。当第一次读取该寄存器时，该位清零，然后才能访问 SPI 数据寄存器	0

3．SPI 数据寄存器 SPDR

SPDR 寄存器用于提供发送和接收的数据字节。处于主模式时，该寄存器将启动 SPI 数据传输。从数据传输开始到 SPIF 状态位置位并且还没有读取状态寄存器的这段时间内不能对该寄存器执行写操作。该寄存器的功能如表 4-29 所列。

表 4-29　SPDR 寄存器功能

位	功　能	说　明	复 位 值
7：0	数据	SPI 双向数据	0

4．SPI 时钟计数寄存器 SPCCR

SPCCR 寄存器控制主机 SCK 的频率。当处于主模式时，SPI 时钟计数器寄存器用于控制传输速率；该寄存器必须在数据传输之前设定。当处于从模式时，该寄存器无效。寄存器的值必须为大于等于 8 的偶数，如果不符合上述条件，可能导致产生不可预知的动作。该寄存器的功能如表 4-30 所列。

表 4-30　SPCCR 寄存器功能

位	功能	说　明	复位值
7：0	计数值	SPI 时钟计数值设定	0

SPI 时钟值计算公式如下：

SPI 时钟值=Fpclk/SPCCR

5．SPI 中断寄存器 SPINT

SPINT 寄存器包含 SPI 接口的中断标志，其功能如表 4-31 所列。

表 4-31　SPINT 寄存器功能

位	功　能	说　明	复 位 值
0	SPI 中断	SPI 中断标志。由 SPI 接口置位以产生中断，向该位写入 1 清零。当 SPIE=1 并且 SPIF 和 WCOL 位中至少有一位为 1 时，该位置位	0
7：1	保留	用户软件不要向其写入 1，读出值未定义	NA

4.4.4 应用举例

使用 SPI 接口之前，要先对引脚进行配置。进行如下设置可将 P0.4、P0.5、P0.6 和 P0.7 选择为 SPI 功能：

```
PINSEL0 = 0x55 << 8;                                          // 可能影响其他引脚功能
PINSEL0 = (PINSEL0 & (~(0xFF << 8))) | (0x55 << 8);           // 不会影响其他引脚功能
```

1．主机操作

下面的步骤描述了 SPI 设置为主机时如何处理数据传输。假设已经设定引脚连接 SPI，并且之前的数据传输已经结束。

1）设置 SPI 时钟计数寄存器 SPCCR，获得期望的 SPI 时钟。

2）设置 SPI 控制寄存器 SPCR，控制 SPI 为主机模式，配置 SPI 时钟极性等。

3）选择从机，将要发送的数据写入 SPI 数据寄存器 SPDR，启动 SPI 数据传输。

4）等待 SPI 状态寄存器 SPSR 中的 SPIF 位置位。SPIF 位将在 SPI 数据传输的最后一个周期之后置位。

5）读取 SPI 状态寄存器 SPSR。

6）从 SPI 数据寄存器 SPDR 中读出接收到的数据（可选）。

7）如果有更多数据需要发送，则跳到第 3 步。

注意，通过读或写 SPI 数据寄存器，可以清零 SPIF 状态位。因此，如果没有读取 SPI 数据寄存器，则需要执行写该寄存器的操作来清零 SPIF 状态位。

2．从机操作

下面的步骤描述了 SPI 设置为从机时如何处理数据传输。假设已经设定引脚连接 SPI，并且之前的数据传输已经结束。要求驱动 SPI 逻辑的系统时钟速度至少 8 倍于 SPI。

1）设置 SPI 控制寄存器 SPCR，控制 SPI 为从机模式，配置 SPI 时钟极性等。

2）将要发送的数据写入 SPI 数据寄存器 SPDR（可选）。注意，这只能在从 SPI 传输没有进行时执行。

3）等待 SPI 状态寄存器 SPSR 中的 SPIF 位置位。SPIF 位将在 SPI 数据传输的最后一个采样钟沿之后置位。

4）读取 SPI 状态寄存器 SPSR。

5）从 SPI 数据寄存器 SPDR 中读出接收到的数据（可选）。

6）如果有更多数据需要发送，则跳到第 2 步。

例 4-7 利用 LPC2138 的 SPI 接口作为主机，对串行存储器 25C160 进行读/写操所。Proteus 仿真电路如图 4-13 所示。25C160 是 SPI 总线的 EEPROM，进行数据写入操作之前，必须先发一个“允许写入锁存”命令，然后再发出“写入”命令和 16 位写入地址，高位在前低位在后。一次最多可写入 16 个字节的数据，注意写入地址安排，不要发生地址翻转。全速运行程序，如果数据读写操作成功，LED1 将点亮；如果出现数据读写错误，LED1 将闪烁。本例对配置文件 config.h 中系统时钟参数设置进行了调整，使得 VPB 时钟为 10MHz，满足 SPI 速率计算公式如下：

$$SPI\ 速率 = VPB\ 时钟/SPCCR = 10MHz/8 <= 2MHz$$

以便能够顺利读/写 25C160。

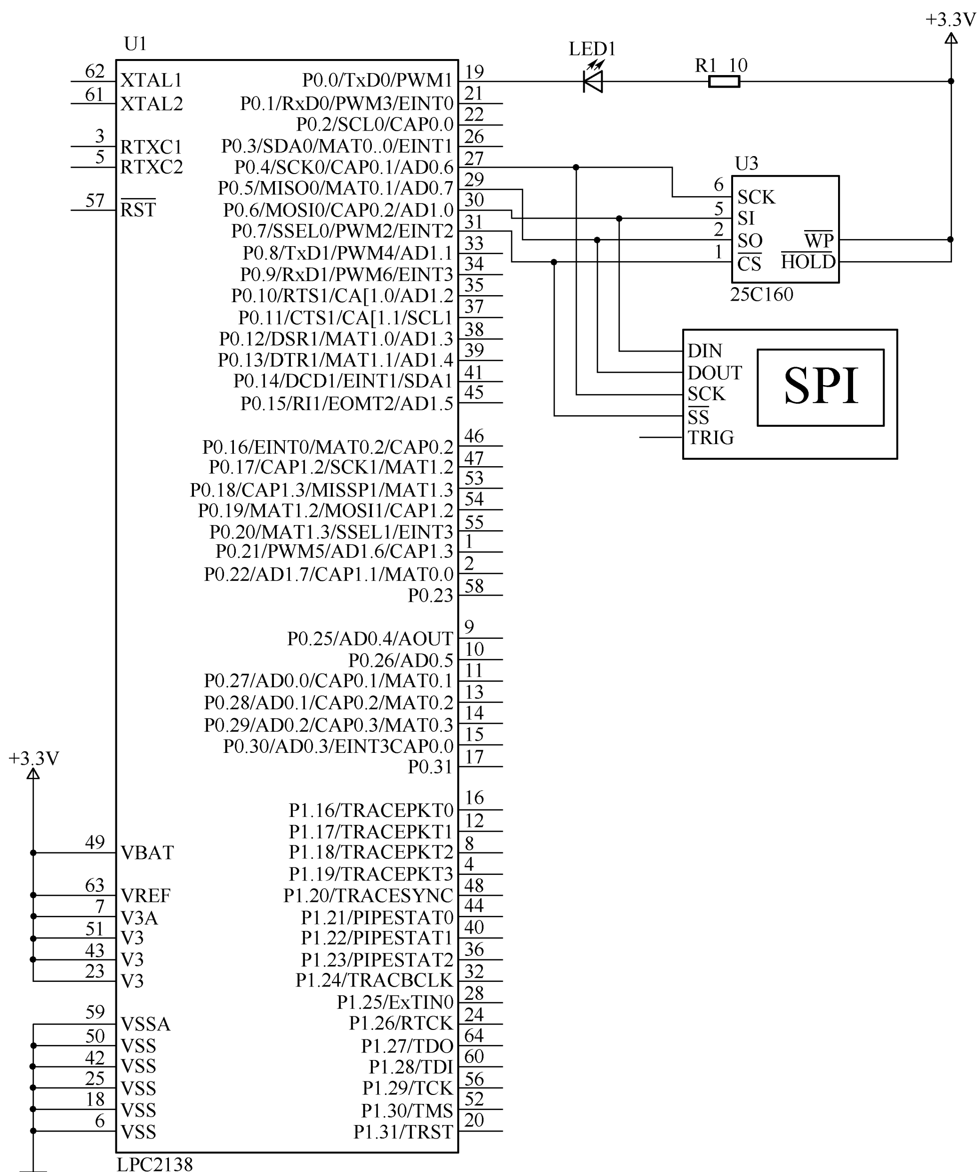


图 4-13　采用 SPI 接口读/写 25C160 的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>
#include "spi.h"

volatile uint32 SPI0Status = 0;
volatile uint32 TxCounter = 0;
uint8 SPICmd[BUFSIZE];
```

```
uint8 SPIWRData[BUFSIZE];
uint8 SPIRDData[BUFSIZE];

/*****************************************************************************
** 函数名称: SPIInit()
** 函数功能：SPI 初始化
*****************************************************************************/
void SPIInit( void )
{    TxCounter = 0;
     S0SPCR = 0x00;
     PINSEL0 &= 0xFFFF00FF;
     PINSEL0 |= 0x00001500;            // SPI 引脚配置
     IODIR0 = SPI0_SEL|LED1;
     IOSET0 = SPI0_SEL|LED1;
     S0SPCCR = 0x8;                    // SPI 时钟配置，SPI 速率= VPB clock/SPCCR value =
                                         10MHz/8 <= 2MHz
     S0SPCR = SPI0_MSTR;
}

/*****************************************************************************
** 函数名称: SPISend()
** 函数功能：向 SPI 端口发送数据块
** 入口参数: buf 数据缓冲区指针， length   数据块长度
*****************************************************************************/
void SPISend( uint8 *buf, uint32 Length )
{    uint32 i;
     uint8 Dummy;
     if ( Length == 0 )  return;
     for ( i = 0; i < Length; i++ )
     {
           S0SPDR = *buf;
           while ( !(S0SPSR & SPIF) );
           Dummy = S0SPDR;
           buf++;
     }
}

/*****************************************************************************
** 函数名称: SPIReceive()
** 函数功能：从 SPI 端口接收数据
** 入口参数: buf 数据缓冲区指针， length   数据块长度
*****************************************************************************/
void SPIReceive( uint8 *buf, uint32 Length )
{     uint32 i;
      for ( i = 0; i < Length; i++ )
      {
```

```
            *buf = SPIReceiveuint8();
            buf++;
        }
}

/*****************************************************************************
** 函数名称: SPIReceiveuint8()
** 函数功能：从 SPI 端口接收单字节数据
** 返回值:    data  接收到的单字节数据
*****************************************************************************/
uint8 SPIReceiveuint8( void )
{       uint8 data;
        S0SPDR = 0xFF;
        while ( !(S0SPSR & SPIF) );              // 等待数据传输结束，SPIF 置位
        data = S0SPDR;
        return ( data );
}

/*****************************************************************************
** 函数名称: main()
** 函数功能：SPI0 配置为主机，对串行 EEPROM 25C160 进行读/写操作
*****************************************************************************/
int main( void )
{       uint32 i, timeout;
        SPIInit();                               // SPI 初始化
        IOCLR0 = SPI0_SEL;
        SPICmd[0] = WREN;                        // EEPROM 允许写入锁存命令
        SPISend( SPICmd, 1 );
        IOSET0 = SPI0_SEL;

    for ( i = 0; i < DELAY_COUNT; i++ );     //  延时至少 250ns

    IOCLR0 = SPI0_SEL;
    SPICmd[0] = RDSR;                            // 检查允许写入锁存状态
    SPISend( SPICmd, 1 );
    SPIReceive( SPIRDData, 1 );
    IOSET0 = SPI0_SEL;
    if ( SPIRDData[0] & (RDSR_WEN|RDSR_RDY) != RDSR_WEN )
    {
              while ( 1 );                       // 允许写入锁存错误
    }

    for ( i = 0; i < BUFSIZE ; i++ )             // 初始化读写缓冲区
    {
        SPIWRData[i+3] = i + 10;                 // 缓冲区头 3 个字节用于 EEPROM 操作命令与地址
```

```
        SPIRDData[i] = 0;
}

IOCLR0 = SPI0_SEL;
SPIWRData[0] = WRITE;                   // 0x02 为 EEPROM 写入命令
SPIWRData[1] = 0x00;                    // 从 0x00 地址开始向 EEPROM 写入 16 个字节数据
SPISend( SPIWRData, BUFSIZE+3 );
IOSET0 = SPI0_SEL;

for ( i = 0; i < DELAY_COUNT; i++ );    // 延时至少 250ns
timeout = 0;
while ( timeout < MAX_TIMEOUT )
{
        IOCLR0 = SPI0_SEL;
        SPICmd[0] = RDSR;               // 检查 EEPROM 写入周期是否结束
        SPISend( SPICmd, 1);
        SPIReceive( SPIRDData, 1 );
        IOSET0 = SPI0_SEL;
        if ( (SPIRDData[0] & RDSR_RDY) == 0x00 )
        {
            break;
        }
        timeout++;
}
if ( timeout == MAX_TIMEOUT )
{
       while( 1 )                       // EEPROM 写入超时
        {
            IOCLR0 = LED1;
            for ( i = 0; i < DELAY_COUNT; i++ );
            IOSET0 = LED1;
        };
}
for ( i = 0; i < DELAY_COUNT; i++ );    // 延时至少 250ns
IOCLR0 = SPI0_SEL;
SPICmd[0] = READ;                       // 0x03 为 EEPROM 读取命令
SPICmd[1] = 0x00;                       // 从 0x00 地址开始读取数据
SPICmd[2] = 0x00;
SPISend( SPICmd, 3 );
SPIReceive( SPIRDData+3, BUFSIZE-3 );
IOSET0 = SPI0_SEL;
for ( i = 3; i < BUFSIZE; i++ )
{
 if ( SPIWRData[i] != SPIRDData[i] )    // 读取数据校验
 {
```

```
            while( 1 )                          // 数据校验出错
            {
                IOCLR0 = LED1;
                for ( i = 0; i < DELAY_COUNT; i++ );
                IOSET0 = LED1;
            };
        }
    }
    while( 1 )
    {
                IOCLR0 = LED1;                  // 数据校验正确
    };
}
```

4.5 SSP 接口

4.5.1 主要特性

LPC2138 具有 1 个可以兼容 SPI、SSI 和 Microwire 总线的 SSP 接口，可以根据应用需要灵活配置。SSP 是一种同步串行接口控制器，可控制 SPI、4 线 SSI 或 Microwire 总线的操作。在一个总线上可以有多个主机或者从机，但是在一次数据传输过程中，总线上只能有一个主机和一个从机进行通信。数据传输原则上是全双工的，4～16 位帧的数据由主机发送到从机或由从机发送到主机。但实际上，大多数情况下只有一个方向上的数据流包含有意义的数据。

SPS 接口的主要特性如下：

- 兼容 Motorola SPI、TI 仪器公司的 4 线 SSI 和 National 半导体的 Microwire 总线。
- 同步串行通信。
- 主机或从机操作。
- 8 帧收发 FIFO。
- 每帧 4～16 位。

4.5.2 寄存器描述

SSP 接口包含 9 个寄存器，分别介绍如下。

1．SSP 控制寄存器 0 SSPCR0

SSPCR0 寄存器用于控制 SSP 的基本操作，其功能如表 4-32 所列。

表 4-32　SSPCR0 寄存器功能

位	功　能	说　明	复 位 值
15:8	SCR	串行时钟速率。值为总线上每 1 位的预分频器输出时钟数减 1	0
7	CPHA	时钟输出相位。该位只用于 SPI 模式。该位为 0 时，SSP 控制器在帧传输的第一个时钟跳变上升沿捕获串行数据；该位为 1 时，SSP 控制器在帧传输的第二个时钟跳变沿捕获串行数据	0

（续）

位	功　能	说　明	复 位 值
6	CPOL	时钟输出极性。该位只用于 SPI 模式。该位为 0 时，SSP 控制器使总线时钟在每帧传输之间保持低电平；该位为 1 时，SSP 控制器使总线时钟在每帧传输之间保持高电平	0
5:4	FRF	帧格式。00＝SPI，01＝SSI，10＝Microwire，11＝保留	0
3:0	DSS	数据长度选择。该字段控制每帧传输的位数目：0000 - 0010=保留，0011＝4 位，0111=8 位，…，1111=16 位	0

2．SSP 控制寄存器 1 SSPCR1

SSPCR1 寄存器用于控制 SSP 的工作方式，其功能如表 4-33 所列。

表 4-33　SSPCR1 寄存器功能

位	功　能	说　明	复 位 值
3	SOD	从机输出禁止。该位只与从机模式有关（MS=1），该位为 1 时，禁止 SSP 控制器驱动 MISO	0
2	MS	主机/从机模式。该位为 0（复位后的值）时，SSP 控制器用作总线主机，驱动 SCLK、MOSI 和 SSEL，并接收 MISO 的信息。该位为 1 时，SSP 控制器用作总线从机，驱动 MISO 和接收 SCLK、MOSI 和 SSEL 的信息。该位只能在 SSE 位为 0 时写入（SSE 位为 1 时可同时写 MS 和 SSE）	0
1	SSE	SSP 使能。该位为 1 时，SSP 控制器可与串行总线上的其他器件相互通信。在置位该位前，软件应将合适的控制信息写入其他 SSP 寄存器和中断控制器寄存器	0
0	LBM	回写模式。正常工作模式下该位为 0；该位为 1 时，串行输入脚可用作串行输出脚（MOSI 或 MISO），而不是仅用作串行输入脚（MISO 或 MOSI 分别起作用）	0

3．SSP 数据寄存器 SSPDR

SSPDR 寄存器的功能如表 4-34 所列，软件可将要发送的数据写入该寄存器，或从该寄存器读出接收到的数据。

表 4-34　SSPDR 寄存器功能

位	功　能	说　明	复 位 值
15：0	DATA	写：当状态寄存器的TNF位为1时，表示Tx FIFO未满，软件可将要发送的数据帧写入该寄存器。如果Tx FIFO以前为空，且SSP控制器空闲，则立刻开始发送数据。否则，写入该寄存器的数据要等到所有数据都发送（或接收）完后才能发送。如果数据长度小于16位，软件必须对数据进行调整后再写入该寄存器 读：当状态寄存器的 RNE 位为 1 时，表示 Rx FIFO 不为空，软件可读取该寄存器。读取该寄存器时，SSP 控制器将返回从 Rx FIFO 中读走的最后一个数据。如果数据长度小于 16 位，该字段的数据必须进行调整，低位对齐，高位补零	0

4．SSP 状态寄存器 SSPSR

SSPSR 寄存器的功能如表 4-35 所列，这是一个只读寄存器，反映 SSP 控制器的当前状态。

表 4-35　SSPSR 寄存器功能

位	功　能	说　明	复 位 值
4	BSY	忙，SSP 控制器空闲时该位为 0；当前正在发送/接收一帧数据或 Tx FIFO 不为空时该位为 1	0
3	RFF	接收 FIFO 满。接收 FIFO 满时该位为 1，反之为 0	0

（续）

位	功　能	说　明	复位值
2	RNE	接收 FIFO 不空。接收 FIFO 为空时该位为 0，反之为 1	0
1	TNF	发送 FIFO 未满。Tx FIFO 满时该位为 0，反之为 1	1
0	TFE	发送 FIFO 空。发送 FIFO 为空时该位为 1，反之为 0	1

5．SSP 时钟预分频寄存器 SSPCPSR

SSPCPSR 寄存器的功能如表 4-36 所列，该寄存器控制 VPB 时钟 Fpclk 得到预分频器时钟因子，同时，预分频时钟被 SCR（SSPCR0 的位 15～8）分频后得到位时钟。

表 4-36　SSPCPSR 寄存器功能

位	功　能	说　明	复位值
7：0	CPSDVSR	这是一个 2～254 中的一个偶数。它是 Fpclk 的分频因子，Fpclk 通过分频后得到预分频器输出时钟。位 0 读出时总是为 0	0

预分频器时钟的计算公式如下：

$$预分频器输出时钟 = Fpclk/CPSDVSR$$

位时钟的计算公式如下：

$$位时钟 = Fpclk/CPSDVSR \times (SCR+1)$$

6．SSP 中断屏蔽设置/清零寄存器 SSPIMSC

SSPIMSC 寄存器的功能如表 4-37 所列，该寄存器控制 SSP 控制器 4 个中断条件的使能。

表 4-37　SSPIMSC 寄存器功能

位	功　能	说　明	复位值
3	TXIM	当 Tx FIFO 至少有一半为空时，软件置位该位来使能中断	0
2	RXIM	当 Rx FIFO 至少有一半为满时，软件置位该位来使能中断	0
1	RTM	当出现接收超时条件时，软件置位该位来使能中断。当 Rx FIFO 不为空且在 32 个位时间内既未接收新数据又未从 FIFO 中读出数据，产生接收超时	0
0	RORIM	当接收溢出时，软件置位该位来使能中断，即当 Rx FIFO 满时又完成另一个帧的接收时，该位置位。接收溢出时，新的数据帧会将前面的数据帧覆盖	0

7．SSP 所有中断状态寄存器 SSPRIS

SSPRIS 寄存器的功能如表 4-38 所列，当一个中断条件出现时，该只读寄存器中对应的位置位，与中断是否通过 SSPIMSC 使能无关，

表 4-38　SSPRIS 寄存器功能

位	功　能	说　明	复位值
3	TXRIS	当 Tx FIFO 至少一半为空时，该位置位	1
2	RXRIS	当 Rx FIFO 至少一半为满时，该位置位	0
1	RTRIS	当接收超时条件出现时，该位置位。一旦开始接收数据，接收超时将撤除	0
0	RORRIS	当 Rx FIFO 满而又接收到另一帧数据时，该位置位。此时接收到的新数据帧会将前面的数据帧覆盖	0

8．SSP 屏蔽中断状态寄存器 SSPMIS

SSPMIS 寄存器的功能如表 4-39 所列，当一个中断条件出现且相应的中断在 SSPMIC

中被使能时，对应在该只读寄存器中的位置位。当产生 SSP 中断时，中断服务程序可通过读该寄存器来判断中断源，

表 4-39　SSPMIS 寄存器功能

位	功　能	说　明	复 位 值
3	TXMIS	当 Tx FIFO 至少一半为空且中断被使能时，该位置位	1
2	RXMIS	当 Rx FIFO 至少一半为满且中断被使能时，该位置位	
1	RTMIS	当接收超时条件出现且中断被使能时，该位置位。一旦开始接收数据，接收超时将撤除	
0	RORMIS	当 Rx FIFO 满而又接收到另一帧数据，并且中断被使能时，该位置位	

9．SSP 中断清零寄存器 SSPICR

SSPICR 寄存器的功能如表 4-40 所列，该寄存器是只写寄存器，向对应位写入 1 将清除相应中断。另外两个中断条件可通过写或读相应的 FIFO 来清除，通过清除 SSPIMSC 中对应的位来禁止。

表 4-40　SSPICR 寄存器功能

位	功　能	说　明	复 位 值
1	RTIC	向该位写 1 来清除接收超时中断	NA
0	RORIC	向该位写 1 来清除“Rx FIFO 满时还接收到数据”的中断	NA

4.5.3　应用举例

使用 SSP 接口之前，要先对引脚进行配置。进行如下设置可将 P0.17、P0.18、P0.19 和 P0.20 选择为 SSP 功能：

```
PINSEL1 = 0xAA << 2;                                    // 可能影响其他引脚的功能
PINSEL1 = (PINSEL1 & (~(0xFF << 2))) | (0xAA << 2);    // 不会影响其他引脚功能
```

LPC2138 的 SSP 可配置成多种接口方式的控制器，并且可以设置每帧数据的长度，另外，SSP 控制器还带有多字节收发缓冲器，可以大大提高通信处理的效率。SSP 的总线时钟设置与 SPI 接口的时钟设置有所差别，SPI 接口只需设置 SPCCR 寄存器就能控制主机 SCK 的频率，而 SSP 控制器不但需要设置时钟预分频寄存器（SSPCRSR），还要设置串行时钟速率（SSPCR0[15:8]）。

SSP 控制器可以提供许多状态信息，并根据设置触发中断。与中断设置相关的寄存器有：SSP 中断屏蔽设置/清除寄存器（SPIMSC）、SSP 所有中断状态寄存器（SSPRIS）和 SSP 屏蔽中断状态寄存器（SSPMIS）。其中，SSPRIS 寄存器反映所有的中断状态，在经过和 SPIMSC 寄存器的内容“相与”操作后，结果反映在 SSPMIS 寄存器中。因此，通过设置 SPIMSC 寄存器过滤掉不关心的中断信息，SSPMIS 寄存器可以触发 IRQ 中断。

例 4-8　SSP 作主机控制 74HC595 驱动 LED 显示器，Proteus 仿真电路如图 4-14 所示。将 SSP 设置为主机，通过引线连接 74HC595，控制 7 段数码管轮流显示 0～F 字样。值得注意的是，采用 SPI 作主机，必须使 SSEL0 为高电平；而采用 SSP 作主机只和一个从机通信时，直接用 SSEL1 作从机片选即可。如果用户在程序中允许了回写模式，则 MISO 引脚可以不连接。

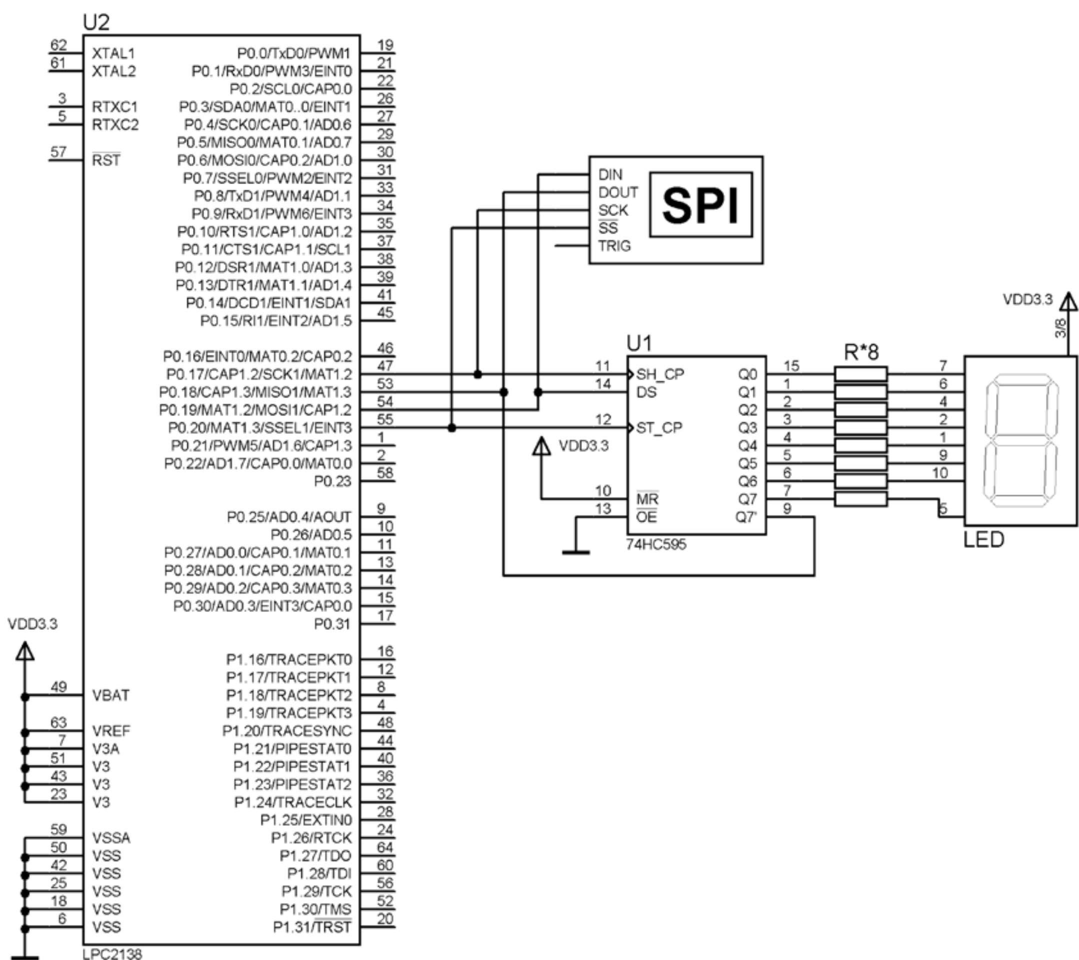


图 4-14　采用 SSP 作主机控制 74HC595 驱动 LED 显示器的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
/****************************************************************************************
** 函数名称：DelayNS()
** 函数功能：长软件延时
** 入口参数：dly 延时参数，值越大，延时越久
****************************************************************************************/
void DelayNS(uint32 dly)
{     uint32 i;
      for(; dly>0; dly--)
            for(i=0; i<50000; i++);
}
```

```
/*********************************************************************************
** 函数名称：SSP_Init()
** 函数功能：将 SSP 控制器设置为主机
*********************************************************************************/
void SSP_Init(void)
{      SSPCR0 = (0x01 << 8) |            // SCR 设置 SPI 时钟分频
                (0x00 << 7) |            // CPHA 时钟输出相位，仅 SPI 模式有效
                (0x01 << 6) |            // CPOL 时钟输出极性，仅 SPI 模式有效
                (0x00 << 4) |            // FRF 帧格式 00=SPI，01=SSI，10=Microwire，11=保留
                (0x07 << 0);             // DSS 数据长度，0000-0010=保留，0011=4 位，0111=8 位，
                                            1111=16 位
           SSPCR1 = (0x00 << 3) |        // SOD 从机输出禁能，1=禁止，0=允许
                (0x00 << 2) |            // MS 主从选择，0=主机，1=从机
                (0x01 << 1) |            // SSE SSP 使能，1=允许 SSP 与其他设备通信
                (0x00 << 0);             // LBM 回写模式
           SSPCPSR = 0x52;               // PCLK 分频值
           SSPIMSC = 0x07;               // 中断屏蔽寄存器
       SSPICR = 0x03;                    // 中断清除寄存器
}

/*********************************************************************************
** 函数名称：SSP_SendData()
** 函数功能：SSP 接口向 SPI 总线发送数据。
** 入口参数：data 待发送的数据
** 出口参数：返回值为读取的数据
*********************************************************************************/
uint8 SSP_SendData(uint8 data)
{
       SSPDR = data;
       while( (SSPSR & 0x01) == 0 );     // 等待 TFE 置位，即发送 FIFO 空
       return(SSPDR);
}

/* LED 数码管字模表 */
uint8 const DISP_TAB[16] = {
           0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8, 0x80,0x90,
           0x88, 0x83, 0xC6, 0xA1,0x86, 0x8E};

/*********************************************************************************
** 函数名称：main
** 函数功能：SSP 作主机，控制 74HC595 驱动 LED 数码管显示
*********************************************************************************/
volatile uint8 rcv;                      // 用于观察读回来的数据

int main(void)
```

```
{     uint8 i;
      PINSEL1 = 0x000002A8;                       // 设置 SSP 引脚连接
      SSP_Init();                                 // 初始化 SSP 接口
      while(1)
      {     for(i=0; i<16; i++)
            {     rcv = SSP_SendData(DISP_TAB[i]);   // 发送显示数据
            DelayNS(10);                          // 延时
            }
      }
}
```

4.6 定时器/计数器

4.6.1 主要特性

LPC2138 具有两个 32 位的可编程定时器/计数器，除了外设基地址不同之外，其他都相同。定时器/计数器对外设时钟 Fpclk 信号进行计数，根据 4 个匹配寄存器的设定，可配置为匹配（即达到匹配寄存器规定的定时值）时产生中断后执行其他动作（输出高/低电平、翻转等）。它还包括 4 个捕获输入，用于在输入信号发生跳变时捕获定时器值，并选择产生中断。

定时器/计数器的主要特性如下：

① 带可编程 32 位预分频器。

② 具有 4 路 32 位的捕获通道，当输入信号跳变时，可捕获定时器的瞬时值，也可选择使捕获事件产生中断。

③ 4 个 32 位匹配寄存器：

- 匹配时定时器继续工作，可选择产生中断。
- 匹配时停止定时器，可选择产生中断。
- 匹配时复位定时器，可选择产生中断。

④ 4 个对应于匹配寄存器的外部输出，具有下列特性：

- 匹配时设置为低电平。
- 匹配时设置为高电平。
- 匹配时翻转。
- 匹配时无动作。

4.6.2 寄存器描述

LPC2138 内部两个定时器/计数器均采用相同的寄存器进行控制，分别介绍如下（其中 x 表示 0 或 1）。

1．中断寄存器 TxIR

TxIR 寄存器包含用于匹配中断和捕获中断的 8 个标志位。如果有中断产生，对应位将置位，否则为 0。向对应位写入 1 将复位中断，写入 0 无效。TxIR 寄存器功能如表 4-41 所列。

表 4-41　TxIR 寄存器功能

位	功　能	说　明	复 位 值
0	MR0 中断	匹配通道 0 的中断标志	0
1	MR1 中断	匹配通道 1 的中断标志	0
2	MR2 中断	匹配通道 2 的中断标志	0
3	MR3 中断	匹配通道 3 的中断标志	0
4	CR0 中断	捕获通道 0 事件的中断标志	0
5	CR1 中断	捕获通道 1 事件的中断标志	0
6	CR2 中断	捕获通道 2 事件的中断标志	0
7	CR3 中断	捕获通道 3 事件的中断标志	0

2．计数控制寄存器 TxCTCR

TxCTCR 寄存器用于选择定时器或计数器模式，以及用于计数的引脚和信号边沿，其功能如表 4-42 所列。

表 4-42　TxCTCR 寄存器功能

位	功　能	说　明	复 位 值
1：0	计数器/定时器模式选择	00：定时器模式，每个Fpclk的上升沿 01：计数器模式，TC在位3：2选择的CAP输入上升沿递增 10：计数器模式，TC在位3：2选择的CAP输入下降沿递增 11：计数器模式，TC 在位 3：2 选择的 CAP 输入上升和下降沿递增	00
3：2	计数器输入选择	当位1：0不是00时，这些位用来选择对哪一个CAP引脚进行采样 00 ＝ CAPn.0 01 ＝ CAPn.1 10 ＝ CAPn.2 11 ＝ CAPn.3 如果在 TnCTCR 中选择计数器模式用于某个特定的 CAPn 输入，则捕获控制寄存器（TnCCR）中对应该输入的 3 位必须设为 000。但是，可在相同的定时器中选择其他 3 个 CAPn 输入用于捕获和/或中断	00

3．定时器控制寄存器 TxTCR

TxTCR 寄存器用于控制定时器的操作，其功能如表 4-43 所列。

表 4-43　TxTCR 寄存器功能

位	功　能	说　明	复 位 值
0	计数器使能	为 1 时，定时器计数器和预分频计数器使能计数；为 0 时，计数器被禁止	0
1	计数器复位	为1时，定时器计数器和预分频计数器在Fpclk的下一个上升沿同步复位；计数器在TCR[1]恢复为0之前保持复位状态	

4．定时器计数器 TxTC

TxTC 为 32 位的定时器计数器。当预分频计数器到达计数的上限时，TC 加 1。如果 TC 在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF，然后翻转到 0x00000000。该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

5．预分频寄存器 TxPR

TxPR 为 32 位的预分频寄存器，用于指定预分频计数器的最大值。分频后的时钟计数频率计算公式如下：

时钟计数频率=Fpclk/(PR+1)

例如当 PR=0 时，定时器计数器每个 Fpclk 周期加 1，当 PR=1 时，定时器计数器每 2 个 Fpclk 周期加 1。

6．预分频计数器 TxPC

TxPC 使用某个常量来控制 Fpclk 的分频，这样可以实现控制定时器分辨率与定时器溢出时间之间的关系。预分频计数器 PC 每个 Fpclk 周期加 1，当其到达预分频寄存器 PR 中保存的值时，定时器计数器 TC 加 1，预分频计数器 PC 在下个 Fpclk 周期复位。

7．匹配寄存器 MR0～MR3

匹配寄存器的值将连续地与定时器计数值相比较，当两个值相等时自动触发相应操作（产生中断、复位定时器计数器或停止定时器），具体执行什么操作由 MCR 寄存器控制。

8．配控制寄存器 TxMCR

TxMCR 寄存器用于控制在发生匹配时所执行的操作，其功能如表 4-44 所列。

表 4-44　MCR 寄存器功能

位	功　能	说　明	复 位 值
0	中断（MR0）	为 1 时，MR0 与 TC 值的匹配将产生中断；为 0 时，中断被禁止	0
1	复位（MR0）	为1时，MR0与TC值的匹配将使TC复位；为0时，该特性被禁止	0
2	停止（MR0）	为1时，MR0与TC值的匹配将使TC和PC停止，TCR[0]清零；为 0 时，该特性被禁止	0
3	中断（MR1）	为1时，MR1与TC值的匹配将产生中断；为0时，中断被禁止	0
4	复位（MR1）	为1时，MR1与TC值的匹配将使TC复位；为0时，该特性被禁止	0
5	停止（MR1）	为1时，MR1与TC值的匹配将使TC和PC 停止，TCR[0]清零；为0时，该特性被禁止	0
6	中断（MR2）	为1时，MR2与TC值的匹配将产生中断；为0时，中断被禁止	0
7	复位（MR2）	为1时，MR2与TC值的匹配将使TC复位；为0时，该特性被禁止	0
8	停止（MR2）	为1时，MR2与TC值的匹配将使TC和PC停止，TCR[0]清零；为0时，该特性被禁止	0
9	中断（MR3）	为1时，MR3与TC值的匹配将产生中断；为0时，中断被禁止	0
10	复位（MR3）	为1时，MR3与TC值的匹配将使TC复位；为0时，该特性被禁止	0
11	停止（MR3）	为1时，MR3与TC值的匹配将使TC和PC停止，TCR[0]清零；为0时，该特性被禁止	0

9．捕获寄存器 CR0～CR3

每个捕获寄存器都与一个器件引脚相关联。当引脚发生特定事件时，可将定时器计数值装入该寄存器。捕获控制寄存器的设定，决定捕获功能是否使能以及捕获事件在引脚的上升沿、下降沿或是双边沿发生。

10．捕获控制寄存器 TxCCR

当发生捕获事件时，捕获控制寄存器用于控制是否将定时器计数值装入 4 个捕获寄存器之一，以及是否产生中断。同时设置上升沿和下降沿位有效的配置，这样会在双边沿触发捕获事件。捕获控制寄存器功能如表 4-45 所列，其中 n 表示定时器编号 0 或 1。

表 4-45　TxCCR 寄存器功能

位	功　能	说　明	复 位 值
0	CAPn.0 上升沿捕获	为 1 时，CAPn.0 上 0 到 1 的跳变将导致 TC 的内容装入 CR0；为 0 时，该特性被禁止	0
1	CAPn.0 下降沿捕获	为 1 时，CAPn.0 上 1 到 0 的跳变将导致 TC 的内容装入 CR0；为 0 时，该特性被禁止	0
2	CAPn.0 事件中断	为 1 时，CAPn.0 的捕获事件所导致的 CR0 装载将产生一个中断；为 0 时，该特性被禁止	0
3	CAPn.1 上升沿捕获	为 1 时，CAPn.1 上 0 到 1 的跳变将导致 TC 的内容装入 CR1；为 0 时，该特性被禁止	0
4	CAPn.1 下降沿捕获	为 1 时，CAPn.1 上 1 到 0 的跳变将导致 TC 的内容装入 CR1；为 0 时，该特性被禁止	0
5	CAPn.1 事件中断	为 1 时，CAPn.1 的捕获事件所导致的 CR1 装载将产生一个中断。为 0 时，该特性被禁止	0
6	CAPn.2 上升沿捕获	为 1 时，CAPn.2 上 0 到 1 的跳变将导致 TC 的内容装入 CR2；为 0 时，该特性被禁止	0
7	CAPn.2 下降沿捕获	为 1 时，CAPn.2 上 1 到 0 的跳变将导致 TC 的内容装入 CR2；为 0 时，该特性被禁止	0
8	CAPn.2 事件中断	为 1 时，CAPn.2 的捕获事件所导致的 CR2 装载将产生一个中断；为 0 时，该特性被禁止	0
9	CAPn.3 上升沿捕获	为 1 时，CAPn.3 上 0 到 1 的跳变将导致 TC 的内容装入 CR3；为 0 时，该特性被禁止	0
10	CAPn.3 下降沿捕获	为 1 时，CAPn.3 上 1 到 0 的跳变将导致 TC 的内容装入 CR3；为 0 时，该特性被禁止	0
11	CAPn.3 事件中断	为 1 时，CAPn.3 上的捕获事件所导致的 CR3 装载将产生一个中断；为 0 时，该特性被禁止	0

11．外部匹配寄存器 TxEMR

外部匹配寄存器 TxEMR 提供外部匹配引脚 MATn(0-3)的控制和状态，其功能如表 4-46 所列。

表 4-46　TxEMR 寄存器功能

位	功　能	说　明	复 位 值
0	外部匹配0	不管 MAT0.0/MAT1.0 是否连接到引脚，该位都会反映其状态。当 MR0 发生匹配时，该输出可翻转、变为低电平、变为高电平或不执行任何动作。位 EMR[4:5]控制该输出的功能	0
1	外部匹配1	不管MAT0.1/MAT1.1是否连接到引脚，该位都会反映其状态。当MR1发生匹配时，该输出可翻转、变为低电平、变为高电平或不执行任何动作。位EMR[6:7]控制该输出的功能	0
2	外部匹配2	不管MAT0.2/MAT1.2是否连接到引脚，该位都会反映其状态。当MR2发生匹配时，该输出可翻转、变为低电平、变为高电平或不执行任何动作。位EMR[8:9]控制该输出的功能	0
3	外部匹配3	不管MAT0.3/MAT1.3 是否连接到引脚，该位都会反映其状态。当MR3发生匹配时，该输出可翻转、变为低电平、变为高电平或不执行任何动作。位EMR[10:11]控制该输出的功能	0
5：4	外部匹配控制0	决定外部匹配0的功能。表4-47所列为这两个位的编码功能	00
7：6	外部匹配控制1	决定外部匹配1的功能。表4-47所列为这两个位的编码功能	00
9：8	外部匹配控制2	决定外部匹配2的功能。表4-47所列为这两个位的编码功能	00
11：10	外部匹配控制3	决定外部匹配3的功能。表4-47所列为这两个位的编码功能	00

表 4-47　外部匹配控制

EMR[11:10]、EMR[9:8] EMR[7:6]、EMR[5:4]	功　能
00	不执行任何操作
01	将对应的外部匹配输出设置为 0（如果连接到引脚，则输出低电平）
10	将对应的外部匹配输出设置为 1（如果连接到引脚，则输出高电平）
11	使对应的外部匹配输出翻转

图 4-15 所示为基本定时器的寄存器功能框图。32 位定时器 TC 的计数频率由 Fpclk 经过预分频控制器 PR 分频得到，而定时器的启动/停止、计数复位则由控制寄存器 TCR 控制，当有捕获事件或者比较匹配事件发生时，中断寄存器 IR 将设置相关中断标志(因为不是定时器溢出而产生中断，所以图上用虚线连接)，若已在 VIC 中打开中断允许，则会产生中断。当然，预分频控制器 PR 只是控制分频数，而其对应的分频计数器是 PC，但用户无须操作 PC 寄存器。

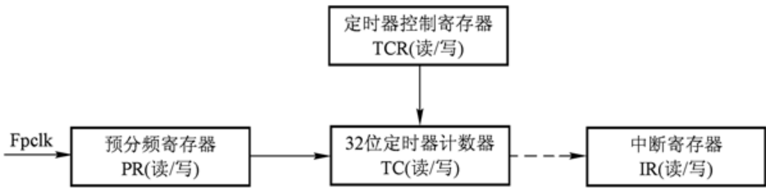


图 4-15　基本定时器的寄存器功能框图

图 4-16 所示为定时器比较匹配功能框图。比较匹配输出由控制寄存器 MCR 进行设置，寄存器 MR0～MR3 为 4 路比较匹配通道的比较值。进行比较匹配时，按照 MCR 的设置产生中断或者复位 TC 等，而外部匹配寄存器 EMR 可以控制比较匹配输出为高电平、低电平、电平翻转等。

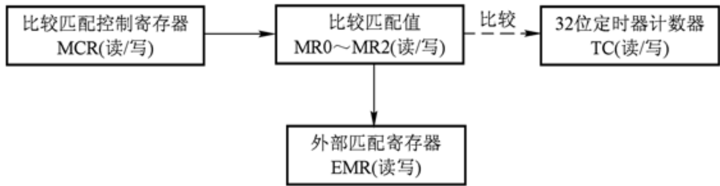


图 4-16　定时器比较匹配功能框图

图 4-17 所示为定时器的捕获寄存器功能框图。定时器 TC 运行过程中，当有捕获触发信号产生时，捕获电路将会立即把当时的定时器 TC 值复制到对应触发通道的捕获寄存器中。可以通过捕获控制寄存器 CCR，将捕获设置为上升沿触发、下降沿触发、双边沿触发，并可设置为捕获中断。

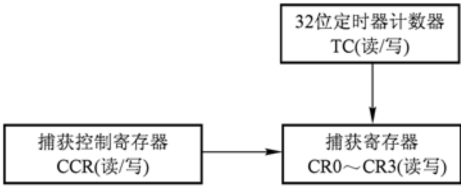


图 4-17　定时器的捕获寄存器功能框图

4.6.3 应用举例

LPC2138 定时器基本操作方法如下：

- 计算定时器的计数时钟频率，设置 PR 寄存器进行分频操作；时钟频率计算公式如下：

定时器计数时钟频率=Fpclk/(N +1)

其中，N 为 PR 的值。

- 设置比较匹配通道的初值及其工作模式，若使用捕获功能，则设置捕获方式。
- 若使用定时器的相关中断，则设置 VIC，使能中断。
- 设置 TCR，启动定时器。

使用定时器功能之前，要先对引脚进行配置。例如，若需要使用捕获或者匹配功能，可进行如下设置，将 P0.12 配置为 MAT1.0 功能：

```
P PINSEL0 = 2 << 24;                                          // 可能会影响其他引脚功能
PINSEL0 = (PINSEL0 & (~(0x03 << 24))) | (0x02 << 24);         // 不会影响其他引脚功能
```

1．定时器定时设置

```
T0MR0 = Fpclk;          // 1s 定时
T0MR0 = Fpclk / 2;      // 0.5s 定时
T0MR0 = Fpclk / 5;      // 0.2s 定时
```

2．定时器匹配时复位并产生中断

```
T0TC = 0;               // 定时器设置为 0
T0PR = 0;               // 时钟不分频
T0MCR = 0x03;           // 设置 T0MR0 匹配后复位 T0TC，并产生中断标志
T0MR0 = Fpclk / 2;      // 0.5s 定时
T0TCR = 0x01;           // 启动定时器
```

3．定时器捕获

```
T1CCR = (1 << 7)|       // 设置 CAP1.2 下降沿捕获
      (1 << 8);         // 允许产生中断
T1TC = 0;               // 定时器设置为 0
T1TCR = 0x01;           // 启动定时器
```

例 4-9 利用定时器 T0 以查询方式实现 0.5s 定时。查询标志位等待定时时间到达，对 LED 进行控制，让 LED 每 0.5s 点亮一次。Proteus 仿真电路如图 4-18 所示。

本例采用 12MHz 晶振，使用 PLL 进行 4 倍频，即 Fcclk = Fosc×4 = 48MHz，外部 VPB 时钟使用复位默认的 Fcclk / 4 = 12MHz；对定时器进行 100 分频（即 PR = 99），T0 的计数时钟频率为 Fpclk/(N+1) = 12MHz/100 = 0.12MHz，计数周期为 8.33μs。要完成 0.5s 定时，则要计数的数值为 500×1000/8.33 = 60024，以匹配通道 0 作为匹配值，匹配时产生中断标志，同时复位 TC。程序查询等待中断标志置位，若定时时间到，先清除中断标志，然后取反 LED 控制口的输出状态。

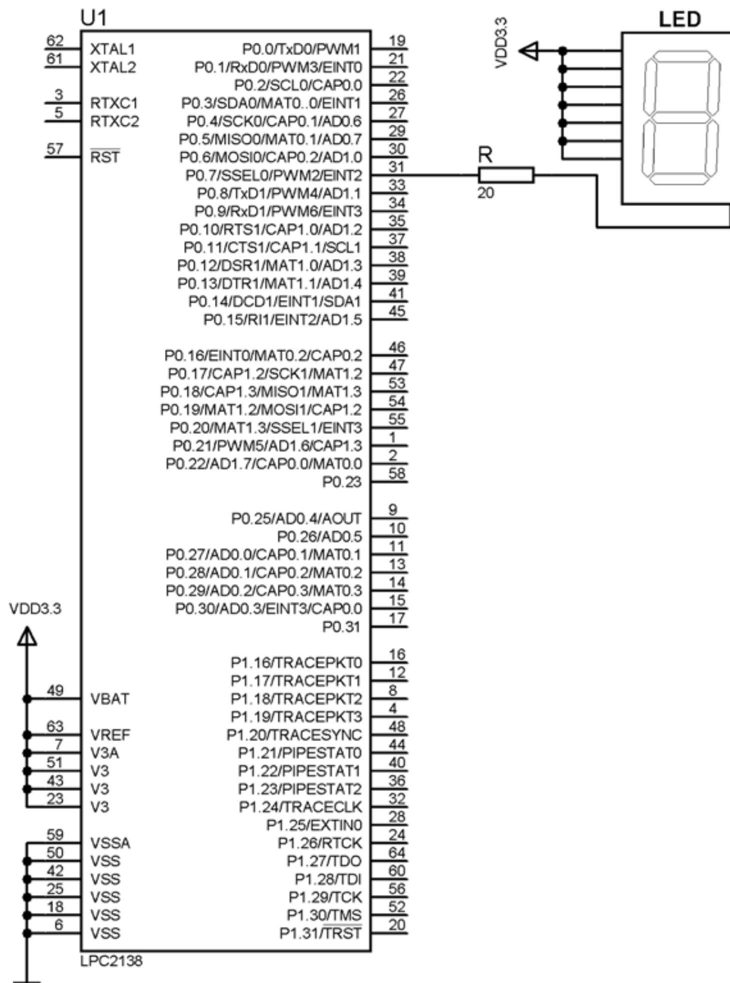


图 4-18　以查询方式实现 0.5s 定时的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include      <intrinsics.h>

#define          LED        (1 << 7)                    /* P0.7 引脚控制 LED，低电平点亮  */

/******************************************************************************
*  函数名称：main()
*  函数功能：通过查询定时器 0 中断标志位控制 LED 点亮
******************************************************************************/
int    main(void)
{
        PINSEL1 = 0x00000000;              //  设置引脚连接 GPIO
```

```
        IO0DIR    = LED;                      // 设置 LED 控制口输出
        T0TC      = 0;                        // 定时器 0 初始化
        T0PR = 99;                            // 设置定时器 0 分频为 100 分频，得 120 000Hz
        T0MCR = 0x03;                         // 匹配通道 0 匹配中断并复位 T0TC
        T0MR0 = 60024;                        // 比较值（0.5s 定时值）
        T0TCR = 0x03;                         // 启动并复位 T0TC
        T0TCR = 0x01;
        while (1)
        {
            while ((T0IR & 0x01) == 0);      // 等待定时时间到
            T0IR = 0x01;                      // 清除中断标志
            if ((IO0SET & LED) == 0)          // 取反 LED 状态
                IO0SET = LED;
            else
                IO0CLR = LED;
        }
}
```

例 4−10　利用定时器 T0 匹配中断方式实现 0.5s 定时。Proteus 仿真电路如图 4−19 所示。

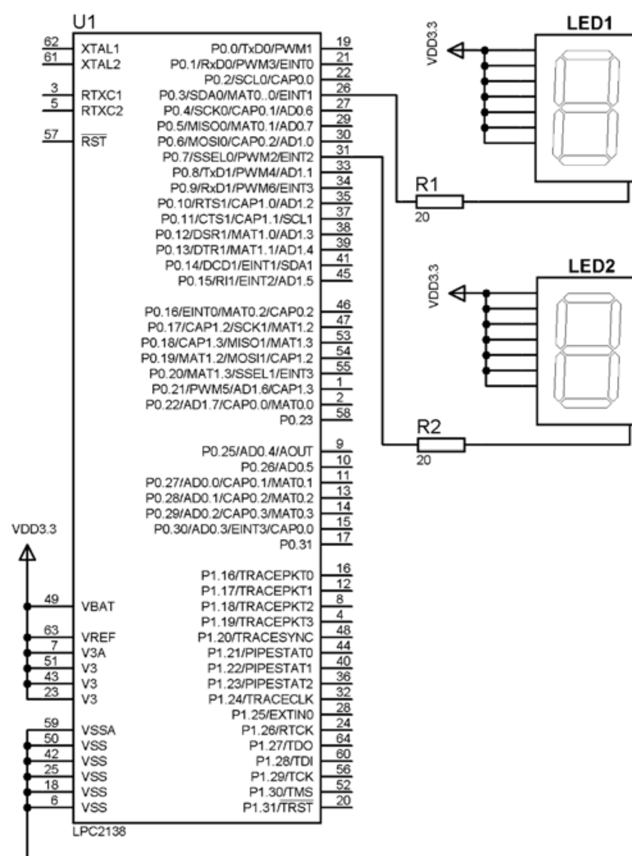


图 4−19　以匹配中断方式实现 0.5s 定时的 Proteus 仿真电路

将引脚 P0.7 连接到 GPIO 控制点亮 LED1，设置 T0 的匹配通道 MR1 匹配中断并复位 TC，在 T0 定时中断函数中对 LED1 状态取反使其点亮或熄灭；将引脚 P0.3 连接到 MAT0.0，设置 T0 的 MR0 匹配输出，输出方式为翻转，控制 LED2 点亮或熄灭。程序运行后两个 LED 同时点亮或熄灭，间隔时间都是 0.5s。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define        LED2    (1 << 7)                // P0.7 引脚控制 LED2，低电平点亮

/*****************************************************************************
** 函数名称 ：IRQ_Timer0()
** 函数功能 ：定时器 0 中断服务程序，取反 LED2 控制口
*****************************************************************************/
   void IRQ_Timer0 (void)
{      uint32 i;
       i=IO0PIN ;
       if ((i & LED2) == 0) IO0SET = LED2;
       else   IO0CLR = LED2;
       T0IR = 0x02;                            // 清除中断标志
       VICVectAddr = 0x00;                     // 通知 VIC 中断处理结束
}

/*****************************************************************************
** 函数名称 ：Time0Init()
** 函数功能 ：定时器 0 初始化
*****************************************************************************/
void   Time0Init(void)
{
       T0TC     = 0;                           // 定时器 T0 初始化
       T0PR = 99;                              // 设置定时器 0 分频为 100 分频，得 120000Hz
       T0MCR = 0x03<<3;                        // 匹配通道 MR1 匹配中断并复位 TC
       T0MR0 = 60024;                          // MR0 比较值（0.5s 定时值）
       T0MR1 = 60024;                          // MR1 比较值（0.5s 定时值）
       T0EMR = 0x30;                           // 设置定时器 0 的 MR0 匹配输出，输出方式为翻转
       T0TCR = 0x03;                           // 启动并复位 T0TC
       T0TCR = 0x01;
       T0IR = 0x02;

       VICIntSelect = 0x00;                    // 所有中断通道设置为 IRQ 中断
       VICVectCntl0 = 0x24;                    // 定时器 0 中断通道分配最高优先级(向量控制器 0)
       VICVectAddr0 = (uint32)IRQ_Timer0;      // 设置中断服务程序地址向量
       VICIntEnable = (1<<4);                  // 使能定时器 0 中断
       __enable_irq();                         // 使能 IRQ 中断
```

```
}

/*****************************************************************************
* 函数名称：main()
* 函数功能：完成引脚配置，调用 T0 初始化，等待中断
*****************************************************************************/
int    main(void)
{
      PINSEL0 = 0x00000080;                    // 设置引脚 P0.3 为 MAT0.0 匹配输出
      PINSEL1 = 0x00000000;                    // 设置引脚连接 GPIO
      IO0DIR    = LED2;                        // 设置 LED2 控制口输出
      Time0Init();                             // 定时器 0 初始化
      IO0CLR = LED2;
      while(1);                                // 等待中断
}
```

例 4-11　利用定时器 T0 的输入捕获功能，获取延时参数，用于控制 LED 的闪烁频率。Proteus 仿真电路如图 4-20 所示。

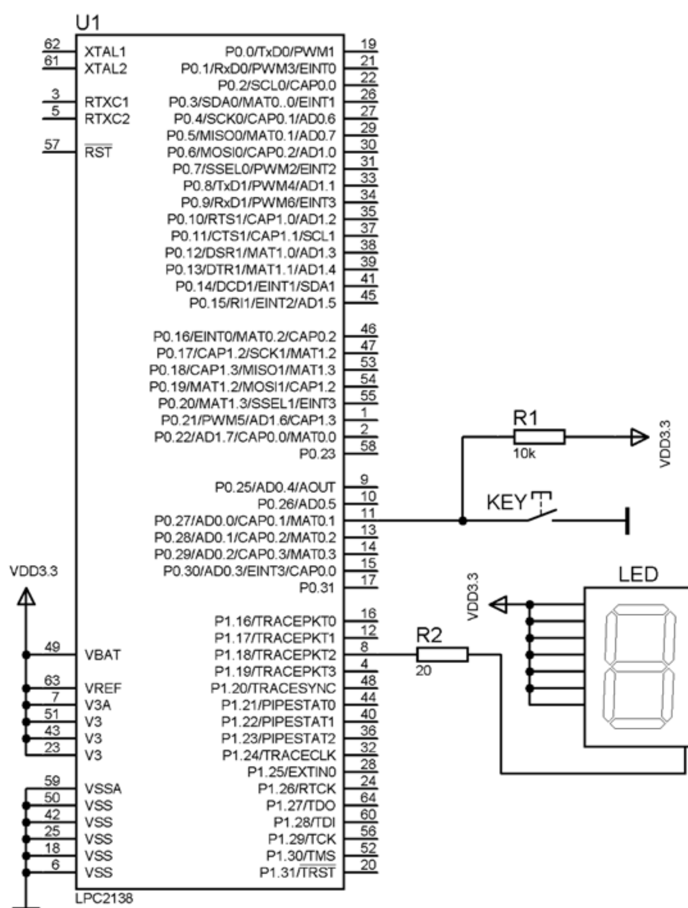


图 4-20　利用输入捕获功能控制 LED 闪烁频率的 Proteus 仿真电路

设置引脚 P0.27 连接到 CAP0.1，并在 P0.27 外部连接按键 KEY，通过 T0CCR 设置 CAP0.1 为下降沿捕获；设置引脚 P1.18 连接到 GPIO 控制点亮 LED；然后启动定时器 0。程序读取按键压下时的捕获值，用作延时参数，控制 LED 闪烁频率。全速运行程序，开始 LED 闪烁频率较快，按一下 KEY 键，LED 闪烁将会变慢，再次按 KEY 键，LED 闪烁将会变得更慢。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

const uint32 LED = 1 << 18;            // P1.18  控制 LED，低电平点亮
/**********************************************************************************
** 函数名称 ：delay()
** 函数功能 ：短软件延时
** 入口参数 ：dly 延时参数，值越大，延时越久
**********************************************************************************/
void delay(uint32 dly)
{       uint32 i;
        for ( ; dly>0; dly--)
                for (i=0; i<10; i++);
}

/*************************************************************************
*  函数名称：main()
*  函数功能：通过按键捕获控制 LED 闪烁延时值
*************************************************************************/
int   main(void)
{       uint32 i=300;
        TargetResetInit();
        PINSEL0=0x00;
        PINSEL1 = (1<<23);          //  设置引脚 P0.27 为 CAP0.1
        IO1DIR = LED;               //  设置 LED 控制口为输出
        T0PR    = 90;
        T0CCR = 1 << 4;             //  设置 CAP0.1 下降沿捕获
        T0TC    = 0;
        T0TCR = 0x01;               //  启动定时器
        IO0CLR=0XFF;
        while (1)
        {
                i = T0CR1;          //  读取捕获到的 T0CR1 值，用作延时参数
                IO1CLR = LED;
                delay(i);
                IO1SET = LED;
                delay(i);
        }
}
```

例 4-12　利用定时器 T0 的计数功能，对按键压下所产生的脉冲信号进行计数，并通过数码管显示脉冲计数值。Proteus 仿真电路如图 4-21 所示。

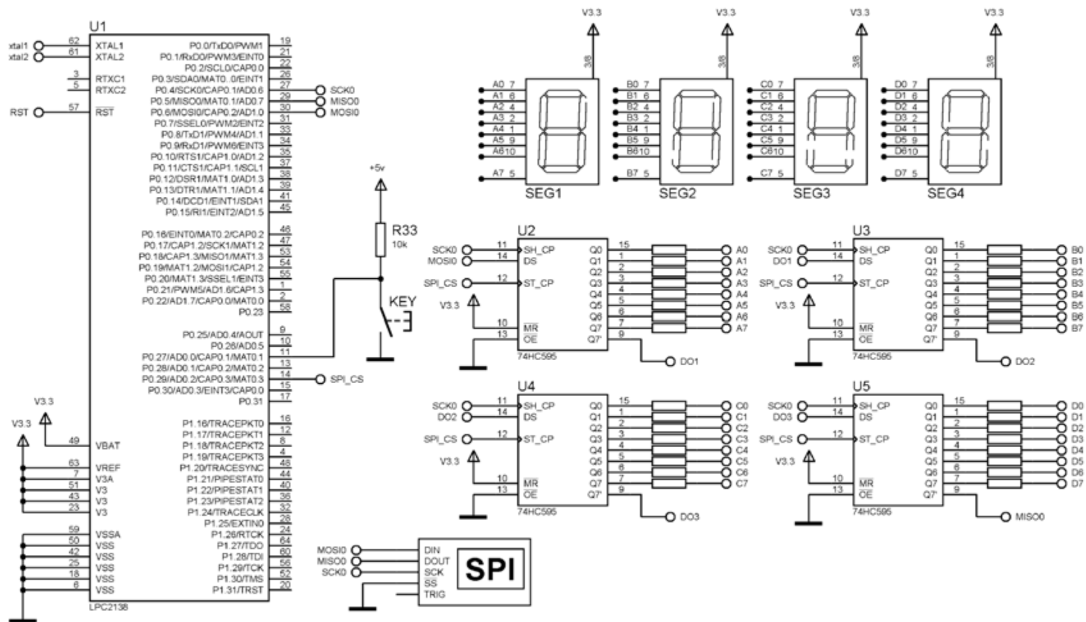


图 4-21　定时器计数功能的 Proteus 仿真电路

本例利用 SPI 接口控制 74HC595 驱动四个数码管，将按键脉冲信号的计数显示出来，每按一次 KEY 键，数码管显示的脉冲数会加 1。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define    HC595_CS         (1<<29)                 // P0.29 口为 74HC595 的片选
/**************************************************************************
*  函数名称：Delayms()
*  函数功能：长软件延时,时间大约为 1ms
*  入口参数：dly  延时参数，值越大，延时越久
**************************************************************************/
void    Delayms(uint32    dly)
{    uint32    i;
     for(; dly>0; dly--)
          for(i=0; i<9000; i++);
}

/**************************************************************************
*  函数名称：MSpiIni()
```

```
* 函数功能：初始化 SPI 接口，设置为主机。
*******************************************************************************/
void  MSpiIni(void)
{
    S0SPCCR = 0x52;                  // 设置 SPI 时钟分频
    S0SPCR = 0x30;                   // 设置 SPI 接口模式，MSTR=1，CPOL=1，CPHA=0，LSBF=0
}

/*******************************************************************************
* 函数名称：MSendData()
* 函数功能：向 SPI 总线发送数据
* 入口参数：data     待发送的数据
* 出口参数：返回值为读取的数据
*******************************************************************************/
uint8  MSendData(uint8 data)
{
    uint8 temp ;
    IO0CLR = HC595_CS;               // 片选
    IO0SET = HC595_CS;
    IO0CLR = HC595_CS;               // 片选
    temp = S0SPDR;
    S0SPDR = data;
    while( 0==(S0SPSR&0x80) );  // 等待 SPIF 置位，即等待数据发送完毕
    IO0SET = HC595_CS;
    return(S0SPDR);
}

/* 此表为 0～F 的字模 */
uint8 const DISP_TAB[16] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,
                               0x88,0x83,0xC6,0xA1,0x86,0x8E
                           };
uint8  rcv_data;
/*******************************************************************************
* 函数名称：main()
* 函数功能：捕获按键 KEY8 按下的次数
*******************************************************************************/
int  main(void)
{
    uint32  i = 2;
    PINSEL0 = 0x00001500;            // 设置 SPI 引脚连接
    PINSEL1 = (1<<23);               // 设置 P0.27 作为 CAP0.1
    IO0DIR = HC595_CS;               // 设置 LED0 控制口为输出
    MSpiIni();                       // 初始化 SPI 接口
    T0PR  = 0;
    T0CCR = 0 << 4;                  // 设置 CAP0.1 下降沿捕获
    T0CTCR = (0x02 << 0) |           // 计数器，在 CAP 的下降沿递增
```

```
                    (0x01 << 2);                                // 选择 CAP0.1
    T0TC    = 0;
    T0TCR = 0x01;                                               // 启动定时器
    while(1)
    {   i = T0TC;                                               // 读取捕获到的 T0TC 值，显示数据
        rcv_data = MSendData(DISP_TAB[i % 10]);                 //  发送显示数据
        rcv_data = MSendData(DISP_TAB[i % 100 / 10]);           //  发送显示数据
        rcv_data = MSendData(DISP_TAB[i % 1000 / 100]);         //  发送显示数据
        rcv_data = MSendData(DISP_TAB[i % 1000 / 10]);          //  发送显示数据
        Delayms(100);
    }
}
```

4.7 脉宽调制器 PWM

4.7.1 主要特性

LPC2138 具有 1 个建立在标准定时器之上的脉宽调制器 PWM，此定时器是 PWM 专用的，不是上一节介绍的定时器 0 或定时器 1。PWM 基于标准的定时器模块，并具有其所有特性，但 LPC2138 只将 PWM 功能输出到引脚。

实际上 PWM 是建立在定时器匹配寄存器基础上的一个附加功能，可独立控制输出波形的上升沿和下降沿位置，从而使 PWM 可应用于更多的领域，如多相位电机控制等。

PWM 的主要特性如下：

① 7 个匹配寄存器，可实现 6 个单边沿控制或 3 个双边沿控制 PWM 输出，或这两种类型的混合输出：

- 连续操作，可选择在匹配时产生中断。
- 匹配时停止定时器，可选择产生中断。
- 匹配时复位定时器，可选择产生中断。

② 每个匹配寄存器对应一个外部输出：

- 匹配时设置为低电平。
- 匹配时设置为高电平。
- 匹配时翻转。
- 匹配时无动作。

③ 支持单边沿控制和/或双边沿控制的 PWM 输出。单边沿控制 PWM 输出在每个周期开始时总是为高电平，除非输出保持恒定低电平。双边沿控制 PWM 输出可在一个周期内的任何位置产生边沿。这样可同时产生正、负脉冲。

④ 脉冲周期和宽度可以是任何的定时器计数值，从而可实现灵活的分辨率和重复速率的设定。所有 PWM 输出都以相同的重复率发生。

⑤ 双边沿控制的 PWM 输出可编程为正、负脉冲。

⑥ 匹配寄存器更新与脉冲输出同步，防止产生错误的脉冲。软件必须在新的匹配值生效之前将它们释放。

⑦ 如果不使能 PWM 模式，则可将其用作一个标准定时器。

⑧ 带可编程 32 位预分频器的 32 位定时器/计数器。

⑨ 当输入信号跳变时，4 个捕获寄存器可取得定时器的瞬时值，也可选择使捕获事件产生中断。

表 4-48 所列为不同 PWM 输出的匹配寄存器选项，其中，“置位”表示输出高电平，“复位”表示输出低电平。支持 N-1 个单边沿 PWM 输出或(N-1)/2 个双边沿 PWM 输出，其中 N 为匹配寄存器的个数。如果需要也可使用混合的 PWM 类型。

表 4-48　不同 PWM 输出的匹配寄存器选项

PWM 通道	单边沿 PWM(PWMSELn=0)		双边沿 PWM(PWMSELn=1)	
	置位	复位	置位	复位
1	匹配 0	匹配 1	匹配 0①	匹配 1①
2	匹配 0	匹配 2	匹配 1	匹配 2
3	匹配 0	匹配 3	匹配 2②	匹配 3②
4	匹配 0	匹配 4	匹配 3	匹配 4
5	匹配 0	匹配 5	匹配 4②	匹配 5②
6	匹配 0	匹配 6	匹配 5	匹配 6

① 这种情况下与单边沿模式相同，因为匹配 0 是相邻的匹配寄存器。基本上 PWM1 不能用作双边沿输出。

② 通常不建议使用 PWM 通道 3 和通道 5 作为双边沿 PWM 输出，因为这样会减少可用的双边沿 PWM 的个数。使用 PWM2、PWM4 和 PWM6 可得到最多个数的双边沿 PWM 输出。

1．单边沿控制的 PWM 输出规则

1）所有单边沿控制的 PWM 输出在 PWM 周期开始时都为高电平，除非它们的匹配值等于 0。

2）每个 PWM 输出在到达其匹配值时都会变为低电平。如果没有发生匹配（即匹配值大于 PWM 速率），PWM 将一直保持高电平。

2．单边沿控制的 PWM 输出规则

当一个新的周期将要开始时，使用以下 5 个规则来决定下一个 PMW 的输出值：

1）在一个 PWM 周期结束时（与下一个 PWM 周期的开始重合的时间点），使用下一个 PMW 周期的匹配值。

2）等于 0 或当前 PWM 速率（与匹配通道 0 的值相同）的匹配值等效。例如，在 PWM 周期开始时的下降沿请求与 PWM 周期结束时的下降沿请求等效。

3）当匹配值正在改变时，如果有其中一个“旧”匹配值等于 PWM 速率，并且新的匹配值不等于 0 或 PWM 速率，若旧的匹配值不等于 0，那么旧的匹配值将再次被使用。

4）如果同时请求 PWM 输出置位和复位，则复位优先。当置位和复位匹配值相同，或者置位或复位值等于 0 并且其他值等于 PWM 速率时，可能发生这种状况。

5）如果匹配值超出范围（大于 PMW 速率值），将不会发生匹配事件，匹配通道对输出不起作用。也就是说，PWM 输出将一直保持同一种状态，可以为低电平、高电平或是“无变化”输出。

4.7.2 寄存器描述

LPC2138 的 PWM 功能由 9 个寄存器进行控制，分别介绍如下。

1．PWM 中断寄存器 PWMIR

PWMIR 寄存器包含 11 个标志位，其中 7 个位用于匹配中断，4 个位保留将来之用。如果有中断产生，PWMIR 中的相应位将置位，否则为 0。向相应的 IR 位写入 1 将复位中断，写入 0 无效。

PWMIR 寄存器功能如表 4-49 所列。

表 4-49　PWMIR 寄存器功能

位	功　　能	说　　明	复　位　值
0	PWMMR0 中断	PWM 匹配通道 0 的中断标志	0
1	PWMMR1 中断	PWM 匹配通道 1 的中断标志	0
2	PWMMR2 中断	PWM 匹配通道 2 的中断标志	0
3	PWMMR3 中断	PWM 匹配通道 3 的中断标志	0
4	保留	应用程序不能向该位写入 1	0
5	保留	应用程序不能向该位写入 1	0
6	保留	应用程序不能向该位写入 1	0
7	保留	应用程序不能向该位写入 1	0
8	PWMMR4 中断	PWM 匹配通道 4 的中断标志	0
9	PWMMR5 中断	PWM 匹配通道 5 的中断标志	0
10	PWMMR6 中断	PWM 匹配通道 6 的中断标志	0

2．PWM 定时控制寄存器 PWMTCR

PWMTC 寄存器用于控制 PWM 定时器计数器的操作，其功能如表 4-50 所列。

表 4-50　PWMTCR 寄存器功能

位	功　　能	说　　明	复　位　值
0	计数器使能	为 1 时，PWM 定时器计数器和 PWM 预分频计数器使能计数；为 0 时，计数器被禁止	0
1	计数器复位	为 1 时，PWM 定时器计数器和 PWM 预分频计数器在 Fpclk 的下一个上升沿同步复位。计数器在 TCR[1]恢复为 0 之前保持复位状态	0
2	保留	应用程序不能向该位写入 1，读出值未定义	NA
3	PWM 使能	为 1 时，PWM 模式使能。PWM 模式将映像寄存器连接到匹配寄存器。只有在 PWMLER 中的相应位置位后发生的匹配 0 事件才会使程序写入匹配寄存器的值生效。需要注意的是，决定 PWM 速率（PWM 匹配 0）的匹配寄存器必须在使能 PWM 之前设定。否则不会发生使映像寄存器内容生效的匹配事件	0

3．PWM 定时器计数器 PWMTC

当预分频计数器到达计数的上限时，32 位定时器计数器加 1。如果 PWMTC 在到达计数上限之前没有被复位，它将一直计数到 0xFFFFFFFF，然后翻转到 0x00000000。该事件不会产生中断。如果需要，可用匹配寄存器检测溢出。

4．PWM 预分频寄存器 PWMPR

32 位预分频寄存器用于计算 PWM 时钟计数频率。PWM 定时器计数器 TC 每经过 PR+1 个 Fpclk 周期加 1。例如，当 PWMPR=0 时，TC 每 1 个 Fpclk 周期加 1，当 PWMPR=1 时，TC 每 2 个 Fpclk 周期加 1。假定 PWMPR 寄存器的值为 PR，则分频后的时钟计数频率计算公式如下：

$$时钟计数频率=Fpclk/(PR+1)$$

5．PWM 预分频计数器 PWMPC

预分频计数器 PWMPC 使用某个常量来控制 Fpclk 的分频，再使之用于 PWM 定时器计数器 TC。预分频计数器 PC 在每个 Fpclk 周期加 1，当其到达 PWM 预分频寄存器中保存的值时，PWM 定时器计数器 TC 加 1，PWM 预分频计数器 PC 在下个 Fpclk 周期复位。

6．PWM 匹配寄存器 PWMMR0～PWMMR6

PWM 匹配寄存器值连续与 PWM 定时器计数器 TC 的值相比较。当两个值相等时自动触发相应动作（产生中断、复位计数器或停止定时器）。具体执行什么动作由 PWMMCR 寄存器控制。

7．PWM 匹配控制寄存器 PWMMCR

PWMMCR 寄存器用于控制在发生匹配时所执行的操作, 其功能如表 4−51 所列。

表 4−51　PWMMCR 寄存器功能

位	功　能	说　明	复 位 值
0	中断（PWMMR0）	为 1 时，PWMMR0 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
1	复位（PWMMR0）	为1时，PWMMR0与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
2	停止（PWMMR0）	为1时，PWMMR0与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0
3	中断（PWMMR1）	为 1 时，PWMMR1 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
4	复位（PWMMR1）	为1时，PWMMR1与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
5	停止（PWMMR1）	为1时，PWMMR2与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0
6	中断（PWMMR2）	为 1 时，PWMMR2 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
7	复位（PWMMR2）	为1时，PWMMR2与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
8	停止（PWMMR2）	为1时，PWMMR2与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0
9	中断（PWMMR3）	为 1 时，PWMMR3 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
10	复位（PWMMR3）	为1时，PWMMR3与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
11	停止（PWMMR3）	为1时，PWMMR3与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0
12	中断（PWMMR4）	为 1 时，PWMMR4 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
13	复位（PWMMR4）	为1时，PWMMR4与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
14	停止（PWMMR4）	为1时，PWMMR4与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0

（续）

位	功　能	说　明	复 位 值
15	中断（PWMMR5）	为1时，PWMMR5 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
16	复位（PWMMR5）	为1时，PWMMR5与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
17	停止（PWMMR5）	为1时，PWMMR5与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0
18	中断（PWMMR6）	为1时，PWMMR6 与 PWMT 值的匹配将产生中断；为 0 时，该中断被禁止	0
19	复位（PWMMR6）	为1时，PWMMR6与PWMTC值的匹配将使PWMTC复位；为0时，该特性被禁止	0
20	停止（PWMMR6）	为1时，PWMMR6与PWMTC值的匹配将使PWMTC和PWMPC停止并使PWMTCR[0]复位为0；为0时，该特性被禁止	0

8．PWM 控制寄存器 PWMPCR

PWMPCR 寄存器用于使能并选择每个 PMW 通道的类型，其功能如表 4-52 所列。

表 4-52　PWMPCR 寄存器功能

位	功　能	说　明	复 位 值
1：0	保留	应用程序不能向该位写入 1，读出值未定义	NA
2	PWMSEL2	为 0 时，PWM2 选择单边沿控制模式；为 1 时，选择双边沿控制模式	0
3	PWMSEL3	为 0 时，PWM3 选择单边沿控制模式；为 1 时，选择双边沿控制模式	0
4	PWMSEL4	为 0 时，PWM4 选择单边沿控制模式；为 1 时，选择双边沿控制模式	0
5	PWMSEL5	为 0 时，PWM5 选择单边沿控制模式；为 1 时，选择双边沿控制模式	0
6	PWMSEL6	为 0 时，PWM6 选择单边沿控制模式；为 1 时，选择双边沿控制模式	0
8：7	保留	应用程序不能向该位写入 1，读出值未定义	NA
9	PWMENA1	为1时，使能PWM1输出；为0时，禁止PWM1输出	0
10	PWMENA2	为1时，使能PWM2输出；为0时，禁止PWM2输出	0
11	PWMENA3	为1时，使能PWM3输出；为0时，禁止PWM3输出	0
12	PWMENA4	为1时，使能PWM4输出；为0时，禁止PWM4输出	0
13	PWMENA5	为1时，使能PWM5输出；为0时，禁止PWM5输出	0
14	PWMENA6	为1时，使能PWM6输出；为0时，禁止PWM6输出	0
15	保留	应用程序不能向该位写入 1，读出值未定义	NA

9．PWM 锁存使能寄存器 PWMLER

当 PWM 匹配寄存器用于产生 PWM 时，PWM 锁存使能寄存器用于控制 PWM 匹配寄存器的更新。当定时器处于 PWM 模式时，如果软件对 PWM 匹配寄存器执行写操作，写入的值将保存在一个映像寄存器中。当 PWM 匹配 0 事件发生时（在 PWM 模式下，通常也会复位定时器），如果对应的锁存使能寄存器位已经置位，那么映像寄存器的内容将传送到实际的匹配寄存器中。此时，新的值将生效并决定下一个 PWM 周期。当发生新值传送时，LER 中的所有位都自动清零。在 PWMLER 中相应位置位和 PWM 匹配 0 事件发生之前，任

何写入 PWM 匹配寄存器的值都不会影响 PWM 操作。

例如，当 PWM2 配置为双边沿操作并处于运行中时，改变定时的典型事件顺序如下：

1）将新值写入 PWM 匹配 1 寄存器。

2）将新值写入 PWM 匹配 2 寄存器。

3）写 PWMLER，同时置位位 1 和位 2。

4）更改的值将在下一次定时器复位时（当 PWM 匹配 0 事件发生时）生效。

写两个 PWM 匹配寄存器的顺序并不重要，因为在写 PWMLER 之前，写入的新匹配值都无效。这样就确保了两个值同时生效。如果使用单个值，也可用同样的方法更改。

PWMLER 中所有位的功能如表 4-53 所列。

表 4-53　PWMLER 寄存器功能

位	功　能	说　明	复 位 值
0	使能PWM匹配0锁存	该位置位将允许最后写入 PWM 匹配 0 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
1	使能PWM匹配1锁存	该位置位将允许最后写入 PWM 匹配 1 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
2	使能PWM匹配2锁存	该位置位将允许最后写入 PWM 匹配 2 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
3	使能PWM匹配3锁存	该位置位将允许最后写入 PWM 匹配 3 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
4	使能PWM匹配4锁存	该位置位将允许最后写入 PWM 匹配 4 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
5	使能PWM匹配5锁存	该位置位将允许最后写入 PWM 匹配 5 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
6	使能PWM匹配6锁存	该位置位将允许最后写入 PWM 匹配 6 寄存器的值在由 PWM 匹配事件引起的下次定时器复位时生效	0
7	保留	应用程序不能向该位写入 1，读出值未定义	NA

LPC2138 的 PWM 功能是建立在标准的定时器之上，它具有 32 位定时器及预分频控制电路，7 个匹配寄存器，可实现 6 个单边 PWM 或 3 个双边 PWM 输出，也可以采用这两种类型的混合输出。具有匹配中断、匹配 PWMTC 复位、匹配 PWMTC 停止功能，如果不使能 PWM 模式，可作为一个标准的定时器。PWM 的基本寄存器功能框图如图 4-22 所示。

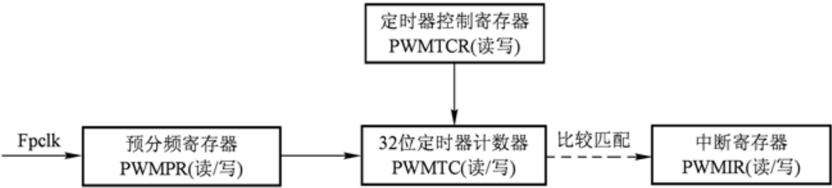


图 4-22　PWM 的基本寄存器功能框图

32 位定时器 PWMTC 的计数频率由 Fpclk 经过 PWMPR 进行分频控制得到，定时器的启动、停止和计数复位由 PWMTCR 控制，当有比较匹配事件发生时，PWMIR 会设置相关中断标志（因为不是定时器溢出而产生中断，所以图 4-22 采用虚线连接），若已打开中断允许（VIC）则会产生中断。当然，预分频控制器 PWMPR 只是控制分频数，而其对应的分

频计数器是 PWMPC，但用户不需要操作 PWMPC 寄存器。

PWM 的比较匹配寄存器功能框图如图 4-23 所示。定时器比较匹配由控制寄存器 PWMMCR 进行匹配操作设置，PWMMR0～6 为 7 路比较匹配通道的比较值寄存器。当比较匹配时，将按照 PWMMCR 设置的方法，产生中断或复位 PWMTC 等，而且 PWMPCR 可以控制单边或双边 PWM 输出，允许或不允许 PWM 输出。另外，为了确保在对 PWMMR0～6 的比较值进行修改过程中不影响 PWM 输出，使用了一个 PWMLER 锁存使能寄存器，需要修改 MR0～6 的比较值时，只有控制 PWMLER 的对应位置位，在匹配 0 事件发生后，此值才会生效。

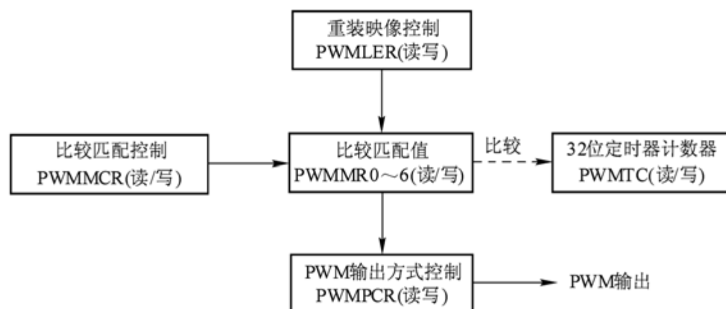


图 4-23　PWM 的比较匹配寄存器功能框图

4.7.3　应用举例

LPC2138 脉宽调制器 PWM 的基本操作方法如下：

- 连接 PWM 功能引脚输出，即设置 PINSEL0、PINSEL1。
- 设置 PWM 定时器的时钟分频值（PWMPR），得到所要的定时器时钟。
- 设置比较匹配控制（PWMMCR），并设置相应的比较值（PWMMRx）。
- 设置 PWM 输出方式并允许 PWM 输出（PWMPCR）及锁存使能控制（PWMLER）。
- 设置 PWMTCR，启动定时器，使能 PWM。
- 如果运行过程中需要更改比较值，更改之后要设置锁存使能。

使用双边沿 PWM 输出时，建议使用 PWM2、PWM4、PWM6；使用单边 PWM 输出时，在 PWM 周期开始时为高电平，匹配后为低电平，使用 PWMMR0 作为 PWM 周期控制，PWMMRx 作为占空比控制。

1．单边沿 PWM 设置

```
PWMPR = 0x00;              //  不分频，计数频率为 Fpclk
PWMMCR = 0x02;             //  设置 PWMMR0  匹配时复位 PWMTC
PWMPCR = 0x0400;           //  允许 PWM2  输出，单边 PWM
PWMMR0 = Fpclk / 1000;     //  设置匹配速率
PWMMR2 = PWMMR0 / 2; // 50%占空比
PWMLER = 0x05;             // PWM0  和 PWM2  匹配锁存
PWMTCR = 0x02;             //  复位 PWMTC
PWMTCR = 0x09;             //  启动 PWM  输出
```

2．双边沿 PWM 设置

```
uint32 tmp;
tmp = Fpclk / 100;
PWMPR = 0x09;              // 对 Fpclk 进行分频，PWM 计数频率为 Fpclk 的 1/10
PWMMCR = 0x02;             // PWMMR0 与 PWMTC 匹配时复位 PWMTC
PWMPCR = (1 << 2) |        // PWM2 双边沿控制
         (1 << 4) |        // PWM4 双边沿控制
         (1 << 6) |        // PWM6 双边沿控制
         (1 << 10) |       // 使能 PWM2 输出
         (1 << 12) |       // 使能 PWM4 输出
         (1 << 14);        // 使能 PWM6 输出
PWMMR0 = tmp;              // PWM 速率控制/计数初值
/* 设置双边沿脉冲的脉宽和位置 */
PWMMR1 = 0;                // PWM2 的位置
PWMMR2 = (tmp / 8) * 7;
PWMMR3 = tmp / 4;          // PWM4 的位置
PWMMR4 = (tmp / 8) * 6;
PWMMR5 = (tmp / 8) * 3;    // PWM6 的位置
PWMMR6 = (tmp / 8) * 5;
```

3．PWM 作为定时器

```
PWMTC = 0;                 // 定时器设置为 0
PWMPR = 0;                 // 时钟不分频
PWMMCR = 0x03;             // 设置 PWMMR0 匹配后复位 PWMTC，并产生中断标志
PWMMR0 = Fpclk / 2;        // 0.5s 定时
PWMTCR = 0x01;             // 启动定时器
```

4．更新 PWM 输出频率

```
PWMMR0 = Fpclk / 10;       // 设置输出频率
PWMLER = 0x05;             // 更新 PWM2 的输出频率。更新匹配值后，必须锁存
```

例 4-13 利用 PWM 作 32 位定时器。虽然 LPC2138 具有两个标准的 32 位定时器，但是在某些特定场合，可能仍不能满足实际的需要，此时，如果不需要使用 PWM 功能，则可以将其作为 32 位定时器使用。Proteus 仿真电路如图 4-24 所示。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

#define     LED (1<<7)
/****************************************************************************
* 函数名称：DelayNS()
* 函数功能：软件延时
```

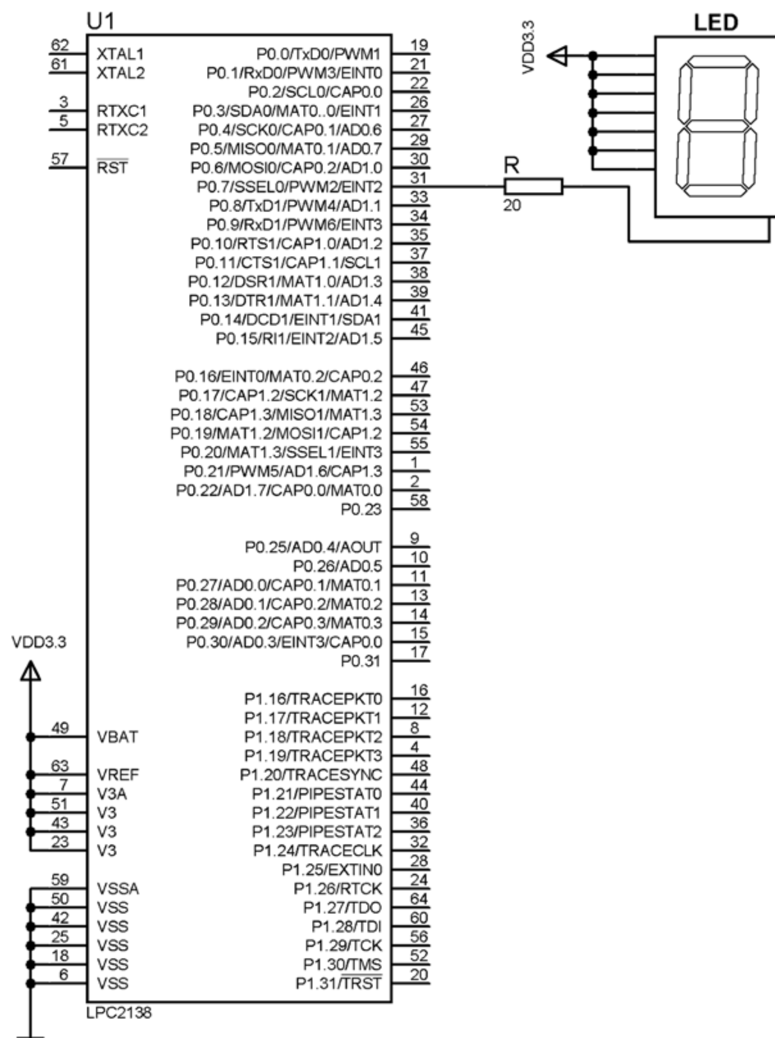


图 4-24　利用 PWM 作 32 位定时器的 Proteus 仿真电路

```
* 入口参数：dly        延时参数，值越大，延时越久
********************************************************************************/
void  DelayNS(uint32  dly)
{
    uint32  i;
    for(; dly>0; dly--)
        for(i=0; i<50000; i++);
}

/*******************************************************************************
** 函数名称: PWM_Int()
** 功能描述: PWM 中断服务函数，取反控制蜂鸣器
********************************************************************************/
```

```
void __irq PWM_Int(void)
{
    PWMIR = 0x01;                               // 复位 PWM 中断标志
    if((IO0SET & LED) == 0) IO0SET = LED;       // 取反 LED 状态
        else IO0CLR = LED;
    VICVectAddr = 0x00;                         // VIC 结束
}

/****************************************************************************
** 函数名称: main()
** 功能描述: 演示 PWM 作 32 位定时器使用实验
****************************************************************************/
int  main(void)
{
   PINSEL0 = 0x00000000;                 // 设置所有引脚连接 GPIO
   PINSEL1 = 0x00000000;                 // 设置引脚连接，P0.20 设置为 EINT0
   IO0DIR=LED;
   IO0CLR=LED;
   PWMTC     = 0;                        // PWM 作为定时器时的初始化，定时器设置为 0
   PWMPR     = 0;                        // 时钟不分频
   PWMMCR   = 0x03;                      // 设置 T0MR0 匹配后复位 T0TC，并产生中断标志
   PWMMR0   = Fpclk/2;                   // 0.5s 定时
   PWMTCR   = 0x01;                      // 启动定时
   __enable_irq();                       // 使能 IRQ 中断

     /* 设置 PWM 中断 */
    VICIntSelect = 0x00000000;           // 所有中断连接 IRQ
    VICVectCntl0 = 0x20|8;               // 分配 PWM 中断到通道 0(最高优先级)
    VICVectAddr0 = (int)PWM_Int;         // PWM 中断服务程序地址
    VICIntEnable = (1 <<8);              // 使能 PWM0 中断
    while(1);                            // 等待中断
}
```

读者可以将本例的代码与前面例 4-10（定时器匹配中断方式）的代码进行比较，可以发现，两者有很多相似之处，说明将 PWM 用作 32 位定时器的操作和普通定时器操作方法一样。

例 4-14 利用 PWM 输出占空比可调节的方波波形。Proteus 仿真电路如图 4-25 所示，每按一次按键 KEY，输出波形的占空比就会发生一次改变。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>

#define    KEY    0x00004000          /*P0.14 引脚连接 KEY*/
```

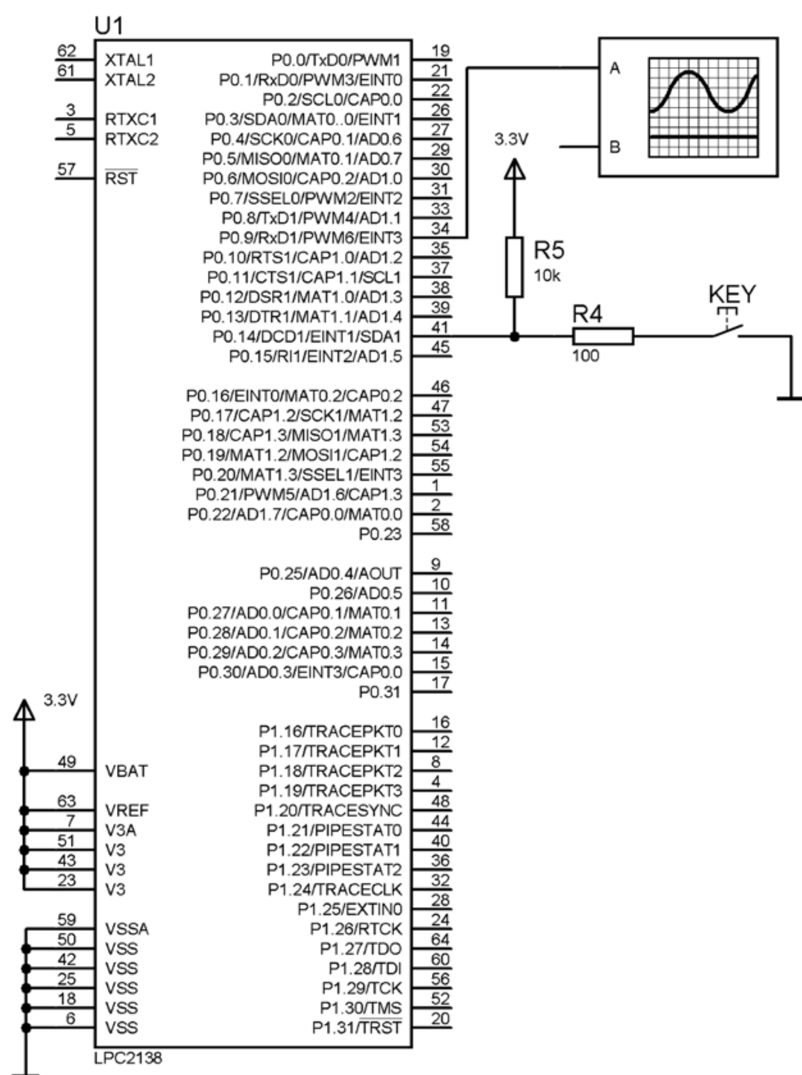


图 4-25　利用 PWM 输出占空比可调波形的 Proteus 仿真电路

```
/*******************************************************************************
*函数名称: WaitKey()
*函数功能: 等待有效按键，有去抖功能
*******************************************************************************/
void WaitKey(void)
{
   uint32 i;
   while(1)
   { while((IO0PIN&KEY)!=0);                          //  等待 KEY 键按下
```

```
        for(i=0;i<50;i++);                          // 延时去抖
        if((IO0PIN&KEY)==0)break;
    }
    while((IO0PIN&KEY)==0);                         // 等待 KEY 键放开
}

/****************************************************************************
*  函数名称：PWM_Init()
*  函数功能：初始化 PWM 功能模块，由 PWM6 输出 PWM 波形
****************************************************************************/
void   PWM_Init(void)
{
      PWMPR=0x00;                                   // 不分频，计数频率为 Fpclk
      PWMMCR=0x02;                                  // 设置 PWMMR0 匹配时复位 PWMTC
      PWMMR0=69120;                                 // 设置 PWM 周期
      PWMMR6=34560;                                 // 设置 PWM 占空比
      PWMLER=0x41;                                  // PWMMR0，PWMMR6 锁存
      PWMPCR=0x4000;                                // 允许 PWM6 输出，单边 PWM
      PWMTCR=0x09;                                  // 启动定时器，PWM 使能
}

/*******************************************************************************
*函数名称: main()
*函数功能: 使用 PWM6 输出占空比可调的 PWM 波形
*******************************************************************************/
int main(void)
{
      uint32   pwmdata;                       // 定义 PWM 占空比控制变量
      PINSEL0=0x00080000;                     // 设置 PWM6 连接到 P0.9 引脚
      PINSEL1=0x00000000;                     // 其他引脚设置为 GPIO
      PWM_Init();                             // PWM 初始化
      pwmdata=34560;
      while(1)
      {     PWMMR0=69120;                     // 设置 PWM 周期
            PWMMR6=pwmdata;                   // 设置 PWM 占空比
            PWMLER=0x41;                      // PWMMR0，PWMMR6 锁存，更新 PWM 占空比
            WaitKey();                        // 等待按键
            pwmdata+=5000;                    // 改变 PWM 占空比控制变量
            if(pwmdata>=69120)pwmdata=0;
      }
}
```

例 4-15　利用 PWM 实现音乐输出。通过改变 PWMMR0 的值，可以改变 PWM 输出频率。本例设定 PWM 输出单边沿 PWM 方波，控制蜂鸣器发声，改变 PWM 的频率，蜂鸣器发出不同频率的声音，实现音乐输出。Proteus 仿真电路如图 4-26 所示。

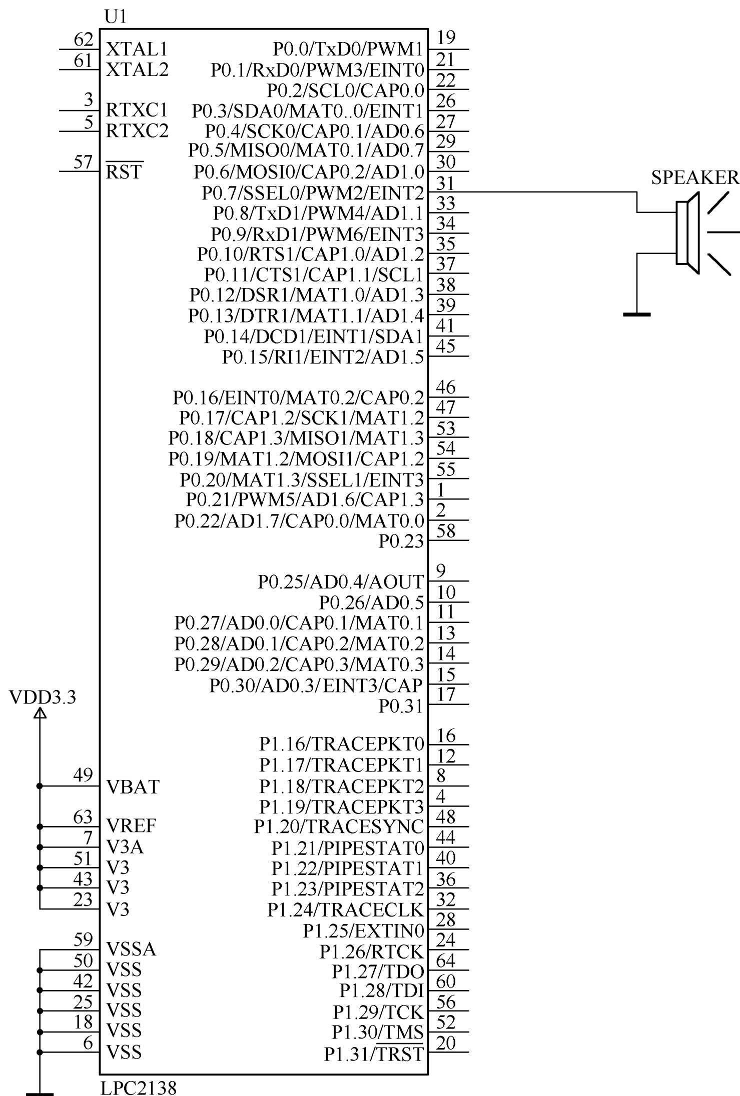


图 4-26　利用 PWM 实现音乐输出的 Proteus 仿真电路

将乐曲简谱和节拍数各自存放于不同数组中，依次从简谱数组中得到相应频率后装入 PWMMR0 中，然后进行锁存，用以改变 PWM 输出频率，并从节拍数组中获得相应的延时参数，两者配合得到音乐输出。

乐曲简谱和节拍头文件 music.h 如下：

```
#ifndef _MUSIC_H
#define _MUSIC_H

// 以 4 分音符为 1 拍
#define TEMPO     8
```

```
#define _1      TEMPO*4     //全音符
#define _1d     TEMPO*6     //附点全音符
#define _2      TEMPO*2     //2 音符
#define _2d     TEMPO*3     //附点 2 音符
#define _4      TEMPO*1     //4 分音符
#define _4d     TEMPO*3/2   //附点 4 分音符
#define _8      TEMPO*1/2   //8 分音符
#define _8d     TEMPO*3/4   //附点 8 音符
#define _16     TEMPO*1/4   //16 分音符
#define _16d    TEMPO*3/8   //附点 16 分音符
#define _32     TEMPO*1/8   //32 分音符

/* 低音 */
#define _1DO    262
#define _1RE    294
#define _1MI    330
#define _1FA    349
#define _1SO    392
#define _1LA    440
#define _1TI    494

/* 中音 */
#define _DO     523
#define _RE     587
#define _MI     659
#define _FA     698
#define _SO     784
#define _LA     880
#define _TI     988

/* 高音 */
#define _DO1    1047
#define _RE1    1175
#define _MI1    1319
#define _FA1    1397
#define _SO1    1568
#define _LA1    1760
#define _TI1    1976
#endif
```

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
```

```
#include "music.h"

/* 歌曲曲谱 */
const uint32 HCMM[] =
{
  _LA, _SO, _MI, _LA, _SO, _MI,
  _LA, _LA, _SO, _LA,
  _LA, _SO, _MI, _LA, _SO, _MI,
  _RE, _RE, _DO, _RE,
  _MI, _MI, _SO, _LA, _DO1, _LA, _SO,
  _MI, _MI, _SO, _DO,
  _MI, _MI, _MI, _MI, _MI,
  _1LA,_1LA,_1SO,_1LA,
};

/* 歌曲节拍 */
const uint32 HCMM_L[] =
{
  _4, _8, _8, _4, _8, _8,
  _8, _4, _8, _2,
  _4, _8, _8, _4, _8, _8,
  _8, _4, _8, _2,
  _4, _8, _8, _8, _8, _8, _8,
  _8, _4, _8, _2,
  _4, _4, _4, _8, _8,
  _8, _4, _8, _2,
};

/*************************************************************************************
** 函数名称: Delay
** 功能描述: 软件延时函数
*************************************************************************************/
void Delay(uint8 dly)
{       uint32 i;
        for(; dly > 0; dly--)
            for(i = 0; i < 0x7FFF; i++);
}

/************************************************************************************
** 函数名称 ：main()
** 函数功能 ：PWM 音乐输出实验
** 调试说明 ：需要将 music.h 包含进来
************************************************************************************/
int main (void)
{     uint8 i;
```

```
    PINSEL0   = 0x02 << 14;                    // P0.7 选择 PWM2 功能

    /* PWM 初始化 */
    PWMPR     = 0x00000000;                    // 不分频，计数频率为 Fpclk
    PWMMCR    = 0x00000002;                    // 设置 PWMMR0 匹配时复位 PWMTC
    PWMPCR    = 0x00000400;                    // 允许 PWM2 输出，单边 PWM
    PWMMR0    = (60000000/4 )/ 1000;
    PWMMR2    = PWMMR0 / 2;                    // 50%占空比
    PWMLER    = 0x00000005;                    // PWM0 和 PWM2 匹配锁存
    PWMTCR    = 0x00000002;                    // 复位 PWMTC
    PWMTCR    = 0x00000009;                    // 启动 PWM 输出

    while(1)
    {
        for(i = 0; i < sizeof(HCMM); i++)
        {
            PWMMR0   = (60000000/4 ) / HCMM[i];     // 设置输出频率
            PWMMR2   = PWMMR0 / 2;
            PWMLER   = 0x05;                        // 更新匹配值后，必须锁存
            Delay(HCMM_L[i]);                       // 延时，控制播放速度
        }
    }
}
```

4.8 A-D 转换器

4.8.1 主要特性

LPC2138 具有两个 10 位逐次逼近式模数转换器。启动 A-D 转换的方式非常灵活，既可以单路软件启动，也可以设置为 BURST 模式对某几路信号逐个循环采样。与其他 LPC2000 系列处理器相比，LPC2138 增加了独立的基准电压源引脚，这对提高转换精度很有利。A-D 转换器的基本时钟由 VPB 总线提供。每个转换器包含一个可编程分频器，可将时钟调整至逐步逼近转换所需的 4.5MHz（最大）；可以完全满足精度要求。

A-D 转换器的主要特性如下：

- 两个 10 位逐次逼近式模数转换器。
- 8 个引脚复用为输入脚。
- 掉电模式。
- 测量范围：0～3V。
- 10 位转换时间≥2.44μs。
- 一个或多个输入的 BURST 转换模式。
- 可选择由输入跳变或定时器匹配信号触发转换。
- 两个转换器的全局启动命令。

4.8.2 寄存器描述

LPC2138 的 A-D 转换器由 3 个寄存器进行控制，分别介绍如下（其中 x 表示 0 或 1）。

（1）A-D 控制寄存器 ADxCR

ADxCR 寄存器控制 A-D 转换通道选择、转换速率、转换精度、起始条件等信息，其功能如表 4−54 所列。

表 4−54　ADxCR 寄存器功能

位	功　能	说　明	复 位 值
7：0	SEL	从AD0.7:0/AD1.7:0引脚中选择采样和转换输入脚。对于AD0，位0选择引脚AD0.0，位7择引脚AD0.7。软件控制模式下，这些位中只有一位可被置位。硬件扫描模式下，SEL的值可为1～8中的任何一个。SEL为零时等效于为0x01	0x01
15：8	CLKDIV	将VPB时钟Fpclk进行（CLKDIV的值+1）分频得到A-D转换时钟，该时钟必须小于或等于4.5MHz。典型的软件将CLKDIV编程为最小值来得到4.5MHz或稍低于4.5MHz的时钟，但某些情况下（例如高阻抗模拟电源）可能需要更低的时钟	0x00
16	BURST	如果该位为0，转换由软件控制，需要11个时钟才能完成转换。如果该位为1，A-D转换器以CLKS字段选择的速率重复执行转换，必要时从SEL字段中为1的位所对应的引脚开始扫描。A-D转换器启动后的第一次转换的是SEL字段中为1的位中的最低有效位对应的模拟输入，然后是为1的更高有效位对应的模拟输入（如果可用）。重复转换通过清零该位终止，但该位被清零时，并不会中止正在进行的转换	0
19：17	CLKS	该字段用来选择BURST模式下每次转换使用的时钟数和所得ADDR转换结果的LS位中可确保精度的位的数目，CLKS可在11个时钟（10位）～4个时钟（3位）之间选择：000＝11个时钟/10 位，001＝10个时钟/9位,…111＝4个时钟/3位	000
21	PDN	1：A-D转换器处于正常工作模式；0：A-D转换器处于掉电模式	0
23：22	TEST[1:0]	这些位用于器件测试。00＝正常模式，01＝数字测试模式，10＝DAC测试模式，11＝一次转换测试模式	00
26：24	START	当BURST为0时，这些位控制A-D转换是否启动和何时启动 000：不启动（PDN清零时使用该值） 001：立即启动转换 010：由EDGE位选择的边沿出现在P0.16/EINT0/MAT0.2/CAP0.2引脚时启动转换 011：由EDGE位选择的边沿出现在P0.22/TD3/CAP0.0/MAT0.0引脚时启动转换 注意：START选择100-111时MAT信号不必输出到引脚上 100：由EDGE位选择的边沿出现在MAT0.1 时启动转换 101：由EDGE位选择的边沿出现在MAT0.3 时启动转换 110：由EDGE位选择的边沿出现在MAT1.0 时启动转换 111：由EDGE位选择的边沿出现在MAT1.1 时启动转换	000
27	EDGE	该位只有在START字段为010～111时有效 0：在所选CAP/MAT信号的下降沿启动转换 1：在所选CAP/MAT信号的上升沿启动转换	0
31：28	保留		0

进入掉电时，PDN 必须在清零 PCONP 的 PCAD0 位前清零，而退出掉电时，PDN 位必须在置位 PCONP 中的 PCAD0 位前置位。

A-D 转换时钟 F_{ADC} 计算公式如下：

$$F_{ADC}=Fpclk/(CLKDIV+1)$$

（2）A-D 数据寄存器 ADxDR

A-D 数据寄存器包含 A-D 转换完成标志和得到的数据等信息，其功能如表 4−55 所列。

表 4-55　ADxDR 寄存器功能

位	功　能	说　明	复 位 值
5：0	保留		0
15：6	V/V3A	当DONE为1时，该字段包含一个二进制数，用来代表SEL字段选中的Ain脚的电压。该字段根据V3A引脚上的电压对Ain脚的电压进行划分。该字段为0表明Ain引脚的电压小于、等于或接近于V_{SSA}；该字段为0x3FF表明Ain引脚的电压接近于、等于或大于V_{3A} 为了测试的需要，写入到该字段的数据捕获到移位寄存器，寄存器的移位时钟为 A-D 转换器时钟。仅当 TEST[1：0]为 10 时，寄存器的 MS 位供给 A-D 转换器的 DINSERI 输入	X
23：16	保留	这些位读出时为 0。用户不应将其置位	0
26：24	CHN	这些位包含的是 LS 位的转换通道	X
29：27	保留	这些位读出时为 0。用户不应将其置位	0
30	OVERUN	BURST 模式下，如果在转换产生 LS 位的结果前一个或多个转换结果被丢失和覆盖，该位置位。在非 FIFO 操作中，该位通过读 ADDR 寄存器清零	0
31	DONE	A-D 转换结束时该位置位。该位在 ADDR 被读出和 ADCR 被写入时清零。如果 ADCR 在转换过程中被写入，该位置位，启动一次新的转换	0

（3）A-D 全局启动寄存器 ADGSR

ADGSR 寄存器用于控制同时启动 2 路 A-D 转换，其功能如表 4-56 所列。

表 4-56　ADGSR 寄存器功能

位	功　能	说　明	复 位 值
15：0	保留		0
16	BURST	如果该位为 0，转换由软件控制，需要 11 个时钟才能完成转换。如果该位为 1，A-D 转换器以 CLKS 字段选择的速率重复执行转换，（如果必要）并从 SEL 字段中为 1 的位对应的引脚开始扫描。A-D 转换器启动后的第一次转换的是 SEL 字段中为 1 的位中的最低有效位对应的模拟输入，然后是为 1 的更高有效位对应的模拟输入（如果可用）。重复转换通过清零该位终止，但该位被清零时并不会中止正在进行的转换	0
23：17	保留		0
26：24	START	当BURST为0 时，这些位控制着A-D转换是否启动和何时启动 000：不启动（PDN 清零时使用该值） 001：立即启动转换 010：由 EDGE 位选择的边沿出现在 P0.16/EINT0/MAT0.2/CAP0.2 引脚时启动转换 011：由EDGE位选择的边沿出现在P0.22/TD3/CAP0.0/ MAT0.0引脚时启动转换 注意：START 选择100～111时MAT 信号不必输出到引脚上 100：由EDGE位选择的边沿出现在MAT0.1时启动转换 101：由EDGE位选择的边沿出现在MAT0.3时启动转换 110：由EDGE位选择的边沿出现在MAT1.0时启动转换 111：寄存器位 27 选择的边沿出现在 MAT1.1 时启动转换	000
27	EDGE	该位只有在START字段为010～111时有效 0：在所选CAP/MAT 信号的下降沿启动转换 1：在所选 CAP/MAT 信号的上升沿启动转换	0
31：28	保留		0

1．硬件触发转换

如果 ADCR 的 BURST 位为 0 且 START 字段的值包含在 010～111 之内，当所选引脚（P0.16 或 P0.22）或定时器匹配信号（MAT0.1、MAT0.3、MAT1.0 或者 MAT1.1）发生跳变时，A-D 转换器启动一次转换。也可选择在 4 个匹配信号中任何一个的指定边沿转换，或者在两个捕获/匹配引脚中任何一个的指定边沿转换。将所选端口的引脚状态或所选的匹配

信号与 ADCR 寄存器的 EDGE 位相异或所得的结果，用作边沿检测逻辑。

2．时钟产生

时钟分频器在 A-D 转换器空闲时保持复位状态，当 ADCR 的 START 字段被写入 01（立即启动转换）或所选边沿出现在选择的信号上时，可立刻启动采样时钟。这个特性可以节省功率，尤其适用于 A-D 转换器很少使用的场合。

3．中断

当 ADCR 寄存器的 DONE 位为 1 时，A-D 转换模块向 VIC 发出中断请求，如果 VIC 中 VICIntEnable 的 A-D 转换中断使能位使能，则会产生中断。读取 ADDR 寄存器将清零 DONE 位。

4．精度和数字接收器

当 A-D 转换器用来测量 Ain 引脚的电压时，可以不理会引脚在引脚选择寄存器中的设置，但是通过禁能引脚的数字接收器来选择 Ain 功能，可以提高转换精度。

当引脚用作 A-D 转换器输入时，不论引脚选择何种功能，它都仍可用作 A-D 输入，A-D 输入可随时被读取，引脚的电压变化可从 A-D 的读取值中反映出来。但是，只有选择模拟输入功能，才能读出有效的模拟值。也只在这种情况下，引脚和 A-D 模块之间的接口电路才有效。其他情况下，执行数字功能所必需的数字逻辑部分将有效，从而影响 A-D 转换器的正确操作。

4.8.3　应用举例

使用 LPC2138 的 A-D 转换器时，先要将测量通道引脚设置为 AINx 功能，然后通过 ADCR 寄存器设置 A-D 转换器的工作模式、转换通道、转换时钟（CLKDIV 时钟分频值），并启动 A-D 转换。可以通过查询或中断方式，检测 A-D 转换是否完成，转换数据保存在 ADDR 寄存器中。

A-D 转换时钟分频值计算公式如下：

$$CLKDIV=Fpclk/Fadclk-1$$

其中 Fadclk 为所要设置的 A-D 转换时钟，其值不能大于 4.5MHz。

1．A-D 转换器初始化

下面是使用 AIN0 进行 10 位 A-D 转换的初始化程序，转换时钟设置为 1MHz。

```
PINSEL1 = 0x00400000;                // 设置 P0.27 为 AIN0 功能
ADCR = (1 << 0) |                    // SEL = 1，选择通道 0
((Fpclk / 1000000 - 1) << 8) |       // CLKDIV = Fpclk / 1000000 – 1，即转换时钟为 1MHz
        (0 << 16) |                  // BURST = 0，软件控制转换操作
        (0 << 17) |                  // CLKS = 0，使用 11clock 转换
        (1 << 21) |                  // PDN = 1，  正常工作模式(非掉电转换模式)
        (0 << 22) |                  // TEST1:0 = 00，正常工作模式(非测试模式)
        (1 << 24) |                  // START = 1，直接启动 A-D 转换
        (0 << 27);                   // 直接启动 A-D 转换，此位无效
```

2．通道切换

进行多通道 A-D 转换的时候，可以按照下面的程序段进行通道切换。

```
ADCR = (ADCR & 0xffffff00 | 0x01 | (1 << 24));          // 切换到通道 1 并进行第一次转换
while ((ADDR & 0x80000000) == 0);                       // 等待转换结束
ADCR = ADCR | (1 << 24);                                // 再次启动转换
while ((ADDR & 0x80000000) == 0);                       // 等待转换结束
ADC_Data = ADDR;                                        // 读取 A-D 转换结果
……
ADCR = (ADCR & 0xffffff00 | 0x02 | (1 << 24));          // 切换到通道 2 并进行第一次转换
while ((ADDR & 0x80000000) == 0);                       // 等待转换结束
ADCR = ADCR | (1 << 24);                                // 再次启动转换
while ((ADDR & 0x80000000) == 0);                       // 等待转换结束
ADC_Data = ADDR;                                        // 读取 A-D 转换结果
……
```

例 4−16　双通道 A-D 转换的实现。设置 P0.27 连接 AD0.0，P0.28 连接 AD0.1，对外部输入电压采样，进行 A-D 转换，转换后的电压值通过 UART0 输出，通过虚拟终端可以看到转换结果。Proteus 仿真电路如图 4−27 所示。

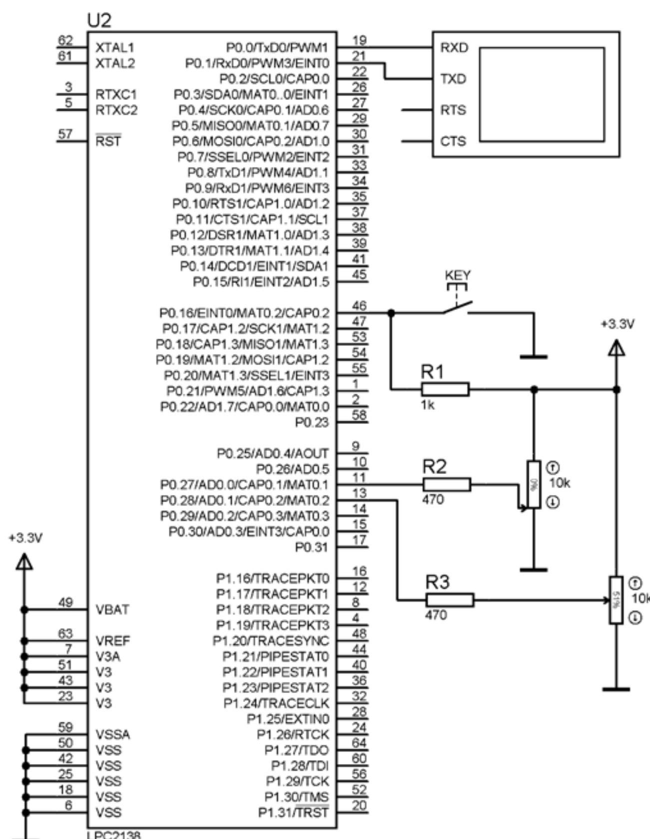


图 4−27　实现 A-D 转换的 Proteus 仿真电路

由于外部输入为 3.30V，且 A-D 转换精度为 10 位，即 2^{10}=1024 级，假定从 A-D 转换寄存器 AD0DR 读到的值为 ADC_Data，则实际电压值可用以下公式计算：

实际电压值=(满额电压/转换精度)×ADC_Data

程序中使用了 printf()函数来输出实际电压值，需要对 Target.c 文件进行适当修改。主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

/****************************************************************************
*  函数名称：DelayNS()
*  函数功能：长软件延时
*  入口参数：dly       延时参数，值越大，延时越久
****************************************************************************/
void   DelayNS(uint32   dly)
{
    uint32   i;
    for(; dly>0; dly--)
        for(i=0; i<5000; i++);
}

/****************************************************************************
*  函数名称：UART0Init()
*  函数功能：初始化串口 0。设置为 8 位数据位，1 位停止位，无奇偶校验
*  入口参数：bps  通信波特率
****************************************************************************/
void   UART0Init(uint32 bps)
{
    uint16 Fdiv;
    PINSEL0 = (PINSEL0 & (~0x0F)) | 0x05;       //  不影响其他引脚连接,设置 I/O 连接到 UART0
    U0LCR = 0x83;                               // DLAB = 1，可设置波特率
    Fdiv = (Fpclk / 16) / bps;                  //  设置波特率
    U0DLM = Fdiv / 256;
    U0DLL = Fdiv % 256;
    U0LCR = 0x03;
}

/****************************************************************************
*  函数名称：putchar()
*  功能描述: 将 printf 函数重新定义到串口 0 输出
****************************************************************************/
int putchar(int data)
{
  while (!(U0LSR & 0x20));
  return (U0THR = data);
}

/****************************************************************************
```

```
* 函数名称：main()
* 函数功能：进行通道 0、1 电压 ADC 转换，并把结果转换成电压值，然后发送到串口
* 说    明：在 CONFIG.H 文件中包含 stdio.h
****************************************************************************/
int  main(void)
{
    uint32  ADC_Data;
    TargetResetInit();
    UART0Init(9600);                        // 初始化 UART0
    PINSEL1 = 0x01400000;                   // 设置 P0.27、P0.28 连接到 AIN0、AIN1
    /* 进行 ADC 模块设置，其中 x<<n 表示第 n 位设置为 x(若 x 超过一位，则向高位顺延) */
    AD0CR = (1 << 0) |                      // SEL = 1，选择通道 0
            ((Fpclk / 1000000 - 1) << 8) |  // 转换时钟为 1MHz
            (0 << 16) |                     // BURST = 0，软件控制转换操作
            (0 << 17) |                     // CLKS = 0，使用 11clock 转换
            (1 << 21) |                     // PDN = 1， 正常工作模式(非掉电转换模式)
            (0 << 22) |                     // TEST1:0 = 00，正常工作模式(非测试模式)
            (1 << 24) |                     // START = 1，直接启动 A-D 转换
            (0 << 27);                  // EDGE = 0 (CAP/MAT 引脚下降沿触发 A-D 转换)
    DelayNS(10);
    ADC_Data = AD0DR;                       // 读取 ADC 结果，并清除 DONE 标志位
    while(1)
    {
        AD0CR = (AD0CR&0x00FFFF00)|0x01|(1 << 24);    // 设置 AD0.0，并进行第一次转换
        while( (AD0DR&0x80000000)==0 );               // 等待转换结束
        AD0CR = AD0CR | (1 << 24);                    // 再次启运转换
        while( (AD0DR&0x80000000)==0 );               // 等待转换结束
        ADC_Data = AD0DR;                             // 读取 ADC 结果
        ADC_Data = (ADC_Data>>6) & 0x3FF;             // 提取 A-D 转换值
        ADC_Data = ADC_Data * 3300/1024;              // 数值转换

        printf("AD0.0=%d mV \r\n", ADC_Data);         // 数据通过串行口输出
        DelayNS(100);

        AD0CR = (AD0CR&0x00FFFF00)|0x02|(1 << 24);    // 设置 AD0.1，并进行第一次转换
        while( (AD0DR&0x80000000)==0 );               // 等待转换结束
        AD0CR = AD0CR | (1 << 24);                    // 再次启运转换
        while( (AD0DR&0x80000000)==0 );               // 等待转换结束
        ADC_Data = AD0DR;                             // 读取 ADC 结果
        ADC_Data = (ADC_Data>>6) & 0x3FF;             // 提取 A-D 转换值
        ADC_Data = ADC_Data * 3300/1024;              // 数值转换

        printf("AD0.1=%d mV \r\r\n", ADC_Data);       // 数据通过串行口输出
        DelayNS(100);
    }

  }
```

例 4-17 硬件触发 A-D 转换。P0.16 引脚的跳变沿可以触发 A-D 转换，当 P0.16 选择 GPIO（输入/输出均可）、EINT0、CAP0.2 时，均可触发 A-D 转换。Proteus 仿真电路如图 4-27 所示。程序运行后，每按一次 KEY 键都将触发 A-D 转换，转换后的电压值通过 UART0 输出，通过虚拟终端可以看到转换结果。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

/****************************************************************************
*  函数名称：DelayNS()
*  函数功能：长软件延时
*  入口参数：dly        延时参数，值越大，延时越久
****************************************************************************/
void   DelayNS(uint32   dly)
{
      uint32   i;
      for(; dly>0; dly--)
            for(i=0; i<5000; i++);
}

/****************************************************************************
*  函数名称：UART0Init()
*  函数功能：初始化串口 0。设置为 8 位数据位，1 位停止位，无奇偶校验
*  入口参数：bps  通信波特率
****************************************************************************/
void   UART0Init(uint32 bps)
{
      uint16 Fdiv;
      PINSEL0 = (PINSEL0 & (~0x0F)) | 0x05;         // 不影响其他引脚连接，设置 I/O 连接到 UART0
      U0LCR = 0x83;                                 // DLAB = 1，可设置波特率
      Fdiv = (Fpclk / 16) / bps;                    // 设置波特率
      U0DLM = Fdiv / 256;
      U0DLL = Fdiv % 256;
      U0LCR = 0x03;
}

/****************************************************************************
*  函数名称：putchar()
*  功能描述: 将 printf 函数重新定义到串口 0 输出
****************************************************************************/
int putchar(int data)
{
   while (!(U0LSR & 0x20));
   return (U0THR = data);
}

/********************************************************************************
```

```
** 函数名称：main()
** 函数功能：进行单通道电压 ADC 转换，并把结果转换成电压值，发送到串口
*******************************************************************************/
int main (void)
{     uint32 ADC_Data;
      PINSEL0 = 0x00000005;          // 引脚连接串口
      PINSEL1 = (1 << 24) | 0x00;    // P0.28 连接到 AD0.1，P0.16 选择 GPIO —可以启动
//    PINSEL1 = (1 << 24) | 0x01;    // P0.28 连接到 AD0.1，P0.16 选择 EINT0 —可以启动
//    PINSEL1 = (1 << 24) | 0x03;    // P0.28 连接到 AD0.1，P0.16 选择 CAP0.2 —可以启动
//    PINSEL1 = (1 << 24) | 2;       // P0.28 连接到 AD0.1，P0.16 连接 MAT0.2
      UART0Init(9600); // 串口初始化
/* 进行 ADC 模块设置 */
      AD0CR = (1 << 1) |                     // SEL=2，选择 AD0.1
            ((Fpclk / 1000000 - 1) << 8) |   // 转换时钟为 1MHz
                  (0 << 16) |                // BURST=0，软件控制转换操作
                  (0 << 17) |                // CLKS=0，使用 11clock 转换
                  (1 << 21) |                // PDN=1，正常工作模式
                  (0 << 22) |                // TEST1:0=00，正常工作模式
                  (2 << 24) |                // START=2，P0.16 引脚边沿启动 A-D 转换
                  (0 << 27);                 // 下降沿
      DelayNS(10);
      ADC_Data = AD0DR;                      // 读取 ADC 结果，并清除 DONE 标志位
      while (1)
      {
            while ((AD0DR & 0x80000000) == 0);        // 等待转换结束
            ADC_Data = AD0DR;                          // 读取 ADC 结果
            ADC_Data = (ADC_Data >> 6) & 0x3ff;
            ADC_Data = ADC_Data * 3300;
            ADC_Data = ADC_Data / 1024;
            printf("AD0.1=%d mV \r\r\n", ADC_Data);    // 数据通过串行口输出
      }
}
```

例 4-18 MAT1.0 匹配触发 A-D 转换。MAT1.0 可以启动 A-D 转换，此时 MAT1.0 信号并不需要输出到相应的引脚上去。Proteus 仿真电路如图 4-27 所示。程序运行后，每当发生 MAT1.0 匹配时都将触发 A-D 转换，转换后的电压值通过 UART0 输出，通过虚拟终端可以看到转换结果。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>

/*****************************************************************************
* 函数名称：DelayNS()
* 函数功能：长软件延时
* 入口参数：dly      延时参数，值越大，延时越久
*****************************************************************************/
```

```
void   DelayNS(uint32   dly)
{
      uint32   i;
      for(; dly>0; dly--)
            for(i=0; i<5000; i++);
}

/****************************************************************************
* 函数名称：UART0Init()
* 函数功能：初始化串口 0。设置为 8 位数据位，1 位停止位，无奇偶校验
* 入口参数：bps  通信波特率
****************************************************************************/
void   UART0Init(uint32 bps)
{
      uint16 Fdiv;
      PINSEL0 = (PINSEL0 & (~0x0F)) | 0x05;        // 不影响其他引脚连接，设置 I/O 连接到 UART0
      U0LCR = 0x83;                                 // DLAB = 1，可设置波特率
      Fdiv = (Fpclk / 16) / bps;                    // 设置波特率
      U0DLM = Fdiv / 256;
      U0DLL = Fdiv % 256;
      U0LCR = 0x03;
}

/****************************************************************************
* 函数名称：putchar()
* 功能描述: 将 printf 函数重新定义到串口 0 输出
****************************************************************************/
int putchar(int data)
{
   while (!(U0LSR & 0x20));
   return (U0THR = data);
}

/*******************************************************************************************
** 函数名称：main()
** 函数功能：MAT1.0 信号启动 A-D 转换
*******************************************************************************************/
int main (void)
{      uint32 ADC_Data;
       PINSEL0 = 0x00000005;                        // 引脚连接串口
       PINSEL1 = 1 << 24;                           // P0.28 连接到 AD0.1
       UART0Init(9600);                             // 串口初始化
/* 设置 MAT1.0 匹配翻转 */
       T1TC   = 0x00;
       T1MCR = 0x02;                                // 设置 T1MR0 匹配后将 T1TC 复位
       T1EMR = 3 << 4;                              // T1MR0 匹配后 MAT1.0 输出翻转
       T1MR0 = Fpclk/8;                             // 0.125s
       T1TCR = 0x03;                                // 启动并复位 T1TC
       T1TCR = 0x01;
```

```
/* 设置 ADC 模块 */
      AD0CR = (1 << 1) |                          // SEL=2，选择通道 AD0.1
              ((Fpclk / 1000000 – 1) << 8) |      // 转换时钟为 1MHz
              (0 << 16) |                         // BURST=0，软件控制转换操作
              (0 << 17) |                         // CLKS=0，使用 11clock 转换
              (1 << 21) |                         // PDN=1，正常工作模式
              (0 << 22) |                         // TEST1:0=00，正常工作模式
              (6 << 24) |                         // START=6，当 MAT1.0 边沿启动 A-D 转换
              (0 << 27);                          // 下降沿
      DelayNS(10);
      ADC_Data = AD0DR;                           // 读取 ADC 结果，并清除 DONE 标志位
      while (1)
      {   while ((AD0DR & 0x80000000) == 0);      // 等待转换结束
          ADC_Data = AD0DR;                           // 读取 ADC 结果
          ADC_Data = (ADC_Data >> 6) & 0x3ff;
          ADC_Data = ADC_Data * 3300;
          ADC_Data = ADC_Data / 1024;
          printf("AD0.1=%d mV \r\r\n", ADC_Data);    // 数据通过串行口输出
      }
}
```

4.9 D-A 转换器

4.9.1 主要特性

LPC2138 具有 1 个 10 位数模转换器，其主要特性如下：

- 10 位数模转换器。
- 电阻串联结构。
- 缓冲输出。
- 掉电模式。
- 选择的转换速率与功率有关。

4.9.2 寄存器描述

LPC2138 的 D-A 转换器只有 1 个 D-A 转换控制寄存器 DACR，该寄存器是一个读/写寄存器，它包含用于模拟转换的数字值和一个用来调节转换性能和功率两者之间的关系的位，其功能如表 4−57 所列。

表 4−57 DACR 寄存器功能

位	功　　能	说　　明	复位值
5：0	保留	用户软件不要向其写入 1，从保留位读出的值未被定义	0x00
15：6	VALUE	当该字段写入一个新值选择好设定时间后，AOUT 引脚上的电压（相对 V_{SSA}）为 VALUE/1024×V_{REF}	0
16	BIAS	该位为 0 时，DAC 的设定时间最大为 1μs，最大电流为 700μA 该位为 1 时，DAC 的设定时间为 2.5μs，最大电流为 350μA	0
31：17	保留	用户软件不要向其写入 1，从保留位读出的值未被定义	0x00

LPC2138 的 D-A 转换器是 10 位 D-A 转换模块，由于 $2^{10}=1024$，因而 VALUE 字段每单位对应的电压值为 $V_{REF}/1024$。如果要得到稳定的输出电压 V_{DA}，则往 VALUE 字段写入的值应为

$$VALUE=(V_{DA}/V_{REF})\times 1024$$

4.9.3 应用举例

LPC2138 的 D-A 转换器使用方法很简单，设定 D-A 引脚连接后往 DACR 寄存器相应字段写入数值即可，如果需要设定 D-A 转换模式，则设定 DACR 寄存器的 BIAS 位即可，然后等待转换完成。实际应用中，对 D-A 的转换时间以及输出电压的性能等要求可能比较苛刻，还需要进行进一步的处理，如参考电压电路设计、D-A 后级处理等。

例 4-19 利用 D-A 转换器产生三角波、锯齿波、正弦波，通过按键进行输出波形切换。Proteus 仿真电路如图 4-28 所示。

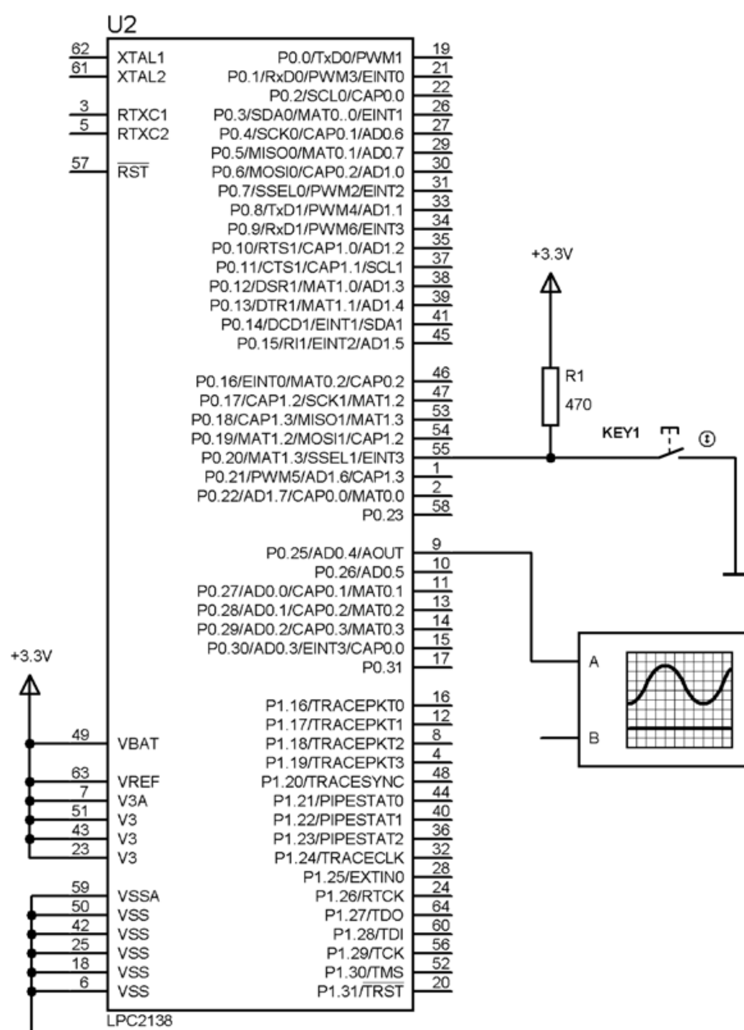


图 4-28 利用 D-A 转换器产生波形的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#define KEY1 (IO0PIN&(1<<20))                    // 定义按键
#define count   sizeof(sindata)/2

uint16 staus=1;
const uint16 sindata[]={                         // 正弦波数据
0x200,0x203,0x206,0x209,0x20c,0x20f,0x212,0x215,0x219,0x21c,0x21f,0x222,
0x225,0x228,0x22b,0x22f,0x232,0x235,0x238,0x23b,0x23e,0x241,0x244,0x247,
0x24b,0x24e,0x251,0x254,0x257,0x25a,0x25d,0x260,0x263,0x266,0x269,0x26d,
0x270,0x273,0x276,0x279,0x27c,0x27f,0x282,0x285,0x288,0x28b,0x28e,0x291,
0x294,0x297,0x29a,0x29d,0x2a0,0x2a3,0x2a6,0x2a9,0x2ac,0x2af,0x2b2,0x2b5,
0x2b8,0x2bb,0x2be,0x2c0,0x2c3,0x2c6,0x2c9,0x2cc,0x2cf,0x2d2,0x2d5,0x2d7,
0x2da,0x2dd,0x2e0,0x2e3,0x2e6,0x2e8,0x2eb,0x2ee,0x2f1,0x2f4,0x2f6,0x2f9,
0x2fc,0x2fe,0x301,0x304,0x307,0x309,0x30c,0x30f,0x311,0x314,0x317,0x319,
0x31c,0x31e,0x321,0x324,0x326,0x329,0x32b,0x32e,0x330,0x333,0x335,0x338,
0x33a,0x33d,0x33f,0x342,0x344,0x347,0x349,0x34b,0x34e,0x350,0x353,0x355,
0x357,0x35a,0x35c,0x35e,0x360,0x363,0x365,0x367,0x369,0x36c,0x36e,0x370,
0x372,0x374,0x376,0x379,0x37b,0x37d,0x37f,0x381,0x383,0x385,0x387,0x389,
0x38b,0x38d,0x38f,0x391,0x393,0x395,0x397,0x399,0x39b,0x39c,0x39e,0x3a0,
0x3a2,0x3a4,0x3a6,0x3a7,0x3a9,0x3ab,0x3ad,0x3ae,0x3b0,0x3b2,0x3b3,0x3b5,
0x3b7,0x3b8,0x3ba,0x3bb,0x3bd,0x3be,0x3c0,0x3c1,0x3c3,0x3c4,0x3c6,0x3c7,
0x3c9,0x3ca,0x3cb,0x3cd,0x3ce,0x3d0,0x3d1,0x3d2,0x3d3,0x3d5,0x3d6,0x3d7,
0x3d8,0x3da,0x3db,0x3dc,0x3dd,0x3de,0x3df,0x3e0,0x3e1,0x3e3,0x3e4,0x3e5,
0x3e6,0x3e7,0x3e7,0x3e8,0x3e9,0x3ea,0x3eb,0x3ec,0x3ed,0x3ee,0x3ef,0x3ef,
0x3f0,0x3f1,0x3f2,0x3f2,0x3f3,0x3f4,0x3f4,0x3f5,0x3f6,0x3f6,0x3f7,0x3f7,
0x3f8,0x3f8,0x3f9,0x3f9,0x3fa,0x3fa,0x3fb,0x3fb,0x3fc,0x3fc,0x3fc,0x3fd,
0x3fd,0x3fd,0x3fe,0x3fe,0x3fe,0x3fe,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,0x3ff,
0x3ff,0x3ff,0x3ff,0x3fe,0x3fe,0x3fe,0x3fe,0x3fd,0x3fd,0x3fd,0x3fc,0x3fc,
0x3fc,0x3fb,0x3fb,0x3fa,0x3fa,0x3fa,0x3f9,0x3f9,0x3f8,0x3f7,0x3f7,0x3f6,
0x3f6,0x3f5,0x3f4,0x3f4,0x3f3,0x3f2,0x3f2,0x3f1,0x3f0,0x3f0,0x3ef,0x3ee,
0x3ed,0x3ec,0x3eb,0x3ea,0x3ea,0x3e9,0x3e8,0x3e7,0x3e6,0x3e5,0x3e4,0x3e3,
0x3e2,0x3e1,0x3e0,0x3de,0x3dd,0x3dc,0x3db,0x3da,0x3d9,0x3d8,0x3d6,0x3d5,
0x3d4,0x3d3,0x3d1,0x3d0,0x3cf,0x3cd,0x3cc,0x3ca,0x3c9,0x3c8,0x3c6,0x3c5,
0x3c3,0x3c2,0x3c0,0x3bf,0x3bd,0x3bc,0x3ba,0x3b9,0x3b7,0x3b5,0x3b4,0x3b2,
0x3b0,0x3af,0x3ad,0x3ab,0x3aa,0x3a8,0x3a6,0x3a4,0x3a2,0x3a1,0x39f,0x39d,
0x39b,0x399,0x397,0x395,0x393,0x392,0x390,0x38e,0x38c,0x38a,0x388,0x386,
0x384,0x382,0x37f,0x37d,0x37b,0x379,0x377,0x375,0x373,0x371,0x36e,0x36c,
0x36a,0x368,0x366,0x363,0x361,0x35f,0x35c,0x35a,0x358,0x355,0x353,0x351,
0x34e,0x34c,0x34a,0x347,0x345,0x342,0x340,0x33d,0x33b,0x339,0x336,0x334,
0x331,0x32e,0x32c,0x329,0x327,0x324,0x322,0x31f,0x31d,0x31a,0x317,0x315,
0x312,0x30f,0x30d,0x30a,0x307,0x305,0x302,0x2ff,0x2fc,0x2fa,0x2f7,0x2f4,
```

```
0x2f1,0x2ef,0x2ec,0x2e9,0x2e6,0x2e4,0x2e1,0x2de,0x2db,0x2d8,0x2d5,0x2d2,
0x2d0,0x2cd,0x2ca,0x2c7,0x2c4,0x2c1,0x2be,0x2bb,0x2b8,0x2b6,0x2b3,0x2b0,
0x2ad,0x2aa,0x2a7,0x2a4,0x2a1,0x29e,0x29b,0x298,0x295,0x292,0x28f,0x28c,
0x289,0x286,0x283,0x280,0x27d,0x27a,0x277,0x273,0x270,0x26d,0x26a,0x267,
0x264,0x261,0x25e,0x25b,0x258,0x255,0x252,0x24e,0x24b,0x248,0x245,0x242,
0x23f,0x23c,0x239,0x236,0x232,0x22f,0x22c,0x229,0x226,0x223,0x220,0x21d,
0x219,0x216,0x213,0x210,0x20d,0x20a,0x207,0x203,0x200,0x1fd,0x1fa,0x1f7,
0x1f4,0x1f1,0x1ed,0x1ea,0x1e7,0x1e4,0x1e1,0x1de,0x1db,0x1d8,0x1d4,0x1d1,
0x1ce,0x1cb,0x1c8,0x1c5,0x1c2,0x1bf,0x1bb,0x1b8,0x1b5,0x1b2,0x1af,0x1ac,
0x1a9,0x1a6,0x1a3,0x1a0,0x19c,0x199,0x196,0x193,0x190,0x18d,0x18a,0x187,
0x184,0x181,0x17e,0x17b,0x178,0x175,0x172,0x16f,0x16c,0x169,0x166,0x163,
0x160,0x15d,0x15a,0x157,0x154,0x151,0x14e,0x14b,0x148,0x145,0x142,0x13f,
0x13c,0x13a,0x137,0x134,0x131,0x12e,0x12b,0x128,0x125,0x123,0x120,0x11d,
0x11a,0x117,0x115,0x112,0x10f,0x10c,0x109,0x107,0x104,0x101,0xff,0xfc,0xf9,
0xf6,0xf4,0xf1,0xee,0xec,0xe9,0xe6,0xe4,0xe1,0xdf,0xdc,0xd9,0xd7,0xd4,0xd2,
0xcf,0xcd,0xca,0xc8,0xc5,0xc3,0xc0,0xbe,0xbb,0xb9,0xb7,0xb4,0xb2,0xaf,0xad,
0xab,0xa8,0xa6,0xa4,0xa1,0x9f,0x9d,0x9b,0x98,0x96,0x94,0x92,0x90,0x8d,0x8b,
0x89,0x87,0x85,0x83,0x81,0x7f,0x7c,0x7a,0x78,0x76,0x74,0x72,0x70,0x6e,0x6d,
0x6b,0x69,0x67,0x65,0x63,0x61,0x5f,0x5e,0x5c,0x5a,0x58,0x56,0x55,0x53,0x51,
0x50,0x4e,0x4c,0x4b,0x49,0x47,0x46,0x44,0x43,0x41,0x3f,0x3e,0x3c,0x3b,0x3a,
0x38,0x37,0x35,0x34,0x32,0x31,0x30,0x2e,0x2d,0x2c,0x2b,0x29,0x28,0x27,0x26,
0x25,0x23,0x22,0x21,0x20,0x1f,0x1e,0x1d,0x1c,0x1b,0x1a,0x19,0x18,0x17,0x16,
0x15,0x14,0x13,0x12,0x12,0x11,0x10,0xf,0xe,0xe,0xd,0xc,0xc,0xb,0xa,0xa,0x9,
0x8,0x8,0x7,0x7,0x6,0x6,0x5,0x5,0x4,0x4,0x3,0x3,0x3,0x2,0x2,0x2,0x1,0x1,0x1,
0x1,0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x1,0x1,0x1,0x1,0x2,0x2,0x2,0x2,0x3,0x3,0x4,0x4,0x4,0x5,0x5,0x6,
0x6,0x7,0x7,0x8,0x8,0x9,0xa,0xa,0xb,0xc,0xc,0xd,0xe,0xf,0xf,0x10,0x11,0x12,
0x13,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,0x1b,0x1c,0x1d,0x1e,0x1f,0x20,
0x21,0x22,0x24,0x25,0x26,0x27,0x28,0x2a,0x2b,0x2c,0x2d,0x2f,0x30,0x31,0x33,
0x34,0x36,0x37,0x38,0x3a,0x3b,0x3d,0x3e,0x40,0x41,0x43,0x44,0x46,0x48,0x49,
0x4b,0x4d,0x4e,0x50,0x52,0x53,0x55,0x57,0x59,0x5a,0x5c,0x5e,0x60,0x62,0x63,
0x65,0x67,0x69,0x6b,0x6d,0x6f,0x71,0x73,0x75,0x77,0x79,0x7b,0x7d,0x7f,0x81,
0x83,0x85,0x87,0x8a,0x8c,0x8e,0x90,0x92,0x94,0x97,0x99,0x9b,0x9d,0xa0,0xa2,
0xa4,0xa7,0xa9,0xab,0xae,0xb0,0xb2,0xb5,0xb7,0xba,0xbc,0xbe,0xc1,0xc3,0xc6,
0xc8,0xcb,0xcd,0xd0,0xd2,0xd5,0xd7,0xda,0xdd,0xdf,0xe2,0xe4,0xe7,0xea,0xec,
0xef,0xf2,0xf4,0xf7,0xfa,0xfc,0xff,0x102,0x105,0x107,0x10a,0x10d,0x110,0x112,
0x115,0x118,0x11b,0x11e,0x120,0x123,0x126,0x129,0x12c,0x12f,0x131,0x134,0x137,
0x13a,0x13d,0x140,0x143,0x146,0x149,0x14c,0x14f,0x152,0x155,0x157,0x15a,0x15d,
0x160,0x163,0x166,0x169,0x16c,0x16f,0x172,0x175,0x178,0x17b,0x17f,0x182,0x185,
0x188,0x18b,0x18e,0x191,0x194,0x197,0x19a,0x19d,0x1a0,0x1a3,0x1a6,0x1a9,0x1ad,
0x1b0,0x1b3,0x1b6,0x1b9,0x1bc,0x1bf,0x1c2,0x1c5,0x1c9,0x1cc,0x1cf,0x1d2,0x1d5,
0x1d8,0x1db,0x1de,0x1e2,0x1e5,0x1e8,0x1eb,0x1ee,0x1f1,0x1f4,0x1f8,0x1fb
};
```

```
/*************************************************************************
* 函数名称：DelayNS()
* 函数功能：长软件延时
* 入口参数：dly        延时参数，值越大，延时越久
*************************************************************************/
void   DelayNS(uint32   dly)
{
    uint32   i;
    for(; dly>0; dly--)
          for(i=0; i<50000; i++);
}

/*************************************************************************
* 函数名称：SINCE()
* 函数功能：产生正弦波
*************************************************************************/
SINCE()
{
    uint16 j,i;
    for(i=0;i<count;i++)
    {
      DACR=(sindata[i]<<6)|(1<<16);    //设定 DAC 的时间为 2.5μs
      for (j=0; j<0x20; j++);          // 等待 D-A 转换完成
    }
}

/*************************************************************************
* 函数名称：JUCHI()
* 函数功能：产生锯齿波
*************************************************************************/
juchi()
{

    uint16 j,i;
    for(i=0;i<1024;i++)
    {
      DACR=(i<<6)|(1<<16);             //设定 DAC 的时间为 2.5μs
      for (j=0; j<0x20; j++);          // 等待 D-A 转换完成
    }
}

/*************************************************************************
* 函数名称：TriAngle()
* 函数功能：产生三角波
```

```
*****************************************************************************/
TriAngle()
{
     uint16 j,i=0;
     for(i=0;i<1024;i++)
      {
        DACR=(i<<6)|(1<<16);                    //  设定 DAC 的时间为 2.5μs
        for (j=0; j<0x20; j++);                 //  等待 D-A 转换完成
    }
    for(i=1023;i>0;i--)
    {
        DACR=(i<<6)|(1<<16);                    //  设定 DAC 的时间为 2.5μs
        for (j=0; j<0x20; j++);                 //  等待 D-A 转换完成
    }
}

/*****************************************************************************
*  函数名称：KEYSCAN()
*  函数功能：按键扫描
*****************************************************************************/
void    KeyScan(void)
{
    if(!KEY1)
    {           DelayNS(10);                    //延时去抖动
                if(!KEY1)
                {
                        staus++;
                        if(staus>2)
                        staus=0;
                }
                 while(!KEY1);                  //等待键松开
    }
}

/*****************************************************************************
*  函数名称：main()
*  函数功能：通过按键控制 D-A 的输出波形
*****************************************************************************/
int    main(void)
{
        PINSEL0 = 0x00000000;                                           //  设置引脚连接
        PINSEL1 = (PINSEL1&(~(0x03<<18))) | (0x02<<18);                 //  P0.25 连接 Aout
        while(1)
         {
```

```
            KeyScan();
            switch(staus)
            {
                case 0:SINCE();break;                           // 产生正弦波
                case 1:TriAngle();break;                        // 产生三角波
                case 2:juchi();break;                           // 产生锯齿波
                default: break;
            }
        }
    }
```

4.10 实时时钟

4.10.1 主要特性

LPC2138 具有 1 个实时时钟（RTC），可以用于进行定时报警、日期、时分秒计时等操作。RTC 消耗的功率非常低，适合于由电池供电、CPU 断续工作的系统。RTC 还具有专用的电源引脚 VBAT，可连接到 3.3V 电池。RTC 时钟信号可由外部独立的 32.768kHz 振荡器或基于 VPB 时钟的可编程预分频器来提供。如果需要使 RTC 中断能够唤醒掉电的 CPU，则必须选择外部时钟源。

实时时钟 RTC 主要特性如下：

- 测量保持日历或时钟的时间通道。
- 超低功耗设计，支持电池供电系统。
- 提供秒、分、小时、日、月、年和星期。
- 可由外部独立的 32kHz 振荡器或片内可编程 VPB 时钟预分频器提供时钟信号。
- 专用电源引脚可与电池或 3.3V 的电压相连。

4.10.2 寄存器描述

RTC 包含许多寄存器，按照功能分成混合寄存器、时间寄存器、时间计数器、报警寄存器和预分频器 4 个组，下面分别予以介绍。

混合寄存器组包含了 5 个用于对 RTC 进行控制的寄存器。

1．中断位置寄存器 ILR

中断位置寄存器 ILR 为 2 位寄存器，由它指定哪些模块可以产生中断。向一个位写入 1，会清除相应的中断，写入 0 无效。读取该寄存器，并将读出的值回写到寄存器中，将会清除检测到的中断。ILR 寄存器的功能如表 4-58 所列。

表 4-58　ILR 寄存器功能

位	功　能	说　明
0	RTCCIF	为 1 时，计数器增量中断模块产生中断；向该位写入 1 清除计数器增量中断
1	RTCALF	为 1 时，报警寄存器产生中断；向该位写入 1 清除报警中断

2．时钟节拍计数器 CTC

时钟节拍计数器 CTC 是一个只读寄存器，它可通过时钟控制寄存器（CCR）复位为 0。CTC 包含时钟分频计数器位，其功能如表 4−59 所列。

表 4−59　CTC 寄存器功能

位	功　能	说　明
0	保留	
15：1	时钟节拍计数器	位于秒计数器之前，CTC 每秒计数 32768 个时钟。由于 RTC 预分频器的关系，这 32768 个时间增量的长度可能并不全部相同

3．时钟控制寄存器 CCR

时钟控制寄存器 CCR 是一个 5 位寄存器，用于控制时钟分频电路的操作，其功能如表 4−60 所列。

表 4−60　CCR 寄存器功能

位	功　能	说　明
0	CLKEN	为 1 时，时间计数器使能；为 0 时，时间计数器都被禁止，这时可对其进行初始化
1	CTCRST	为 1 时，时钟节拍计数器复位。在该位变为 0 之前，它将一直保持复位状态
3:2	CTTEST	正常操作中，这些位应当全 0
4	CLKSRC	为 0 时，时钟节拍计数器计数预分频器的时钟；为 1 时，CTC 计数连接在 RTCX1 和 RTCX2 两端的 32kHz 振荡器信号

只有当 CLKEN 为 0 时才可以对时间计数器（SEC、MIN、HOUR、DOM、DOW、DOY、MONTH 和 YEAR）进行设置。

4．计数器增量中断寄存器 CIIR

时钟控制寄存器 CIIR 可使计数器每次增加时产生一次中断，其功能如表 4−61 所列。

表 4−61　CIIR 寄存器功能

位	功　能	说　明
0	IMSEC	为 1 时，秒值的增加产生一次中断
1	IMMIN	为 1 时，分值的增加产生一次中断
2	IMHOUR	为 1 时，小时值的增加产生一次中断
3	IMDOM	为 1 时，日期（月）值的增加产生一次中断
4	IMDOW	为 1 时，星期值的增加产生一次中断
5	IMDOY	为 1 时，日期（年）值的增加产生一次中断
6	IMMON	为 1 时，月值的增加产生一次中断
7	IMYEAR	为 1 时，年值的增加产生一次中断

5．报警屏蔽寄存器 AMR

报警屏蔽寄存器 AMR 允许用户屏蔽任意报警寄存器。对于报警功能来说，要产生中断，未屏蔽的报警寄存器必须匹配对应的时间计数值。只有当计数器之间的比较第一次从不匹配到匹配时才会产生中断。向中断位置寄存器（ILR）的位 1 写入 1 会清除相应的中断。如果所有屏蔽位都置位，报警将被禁止。AMR 寄存器功能如表 4−62 所列。

表 4-62　AMR 寄存器功能

位	功　能	说　明
0	AMRSEC	为 1 时，秒值不与报警寄存器比较
1	AMRMIN	为 1 时，分值不与报警寄存器比较
2	AMRHOUR	为 1 时，小时值不与报警寄存器比较
3	AMRDOM	为 1 时，日期（月）值不与报警寄存器比较
4	AMRDOW	为 1 时，星期值不与报警寄存器比较
5	AMRDOY	为 1 时，日期（年）值不与报警寄存器比较
6	AMRMON	为 1 时，月值不与报警寄存器比较
7	AMRYEAR	为 1 时，年值不与报警寄存器比较

时间寄存器组包含 3 个完整时间寄存器，只需执行 3 次读操作，即可读出所有时间计数器的值。

1．完整时间寄存器 0 CTIME0

完整时间寄存器 0 包含的时间值为：秒、分、小时和星期，如表 4-63 所列。

表 4-63　CTIME0 寄存器功能

位	功　能	说　明
31:27	保留	
26:24	星期值	该值的范围为 0～6
23:21	保留	
20:16	小时值	该值的范围为 0～23
15:14	保留	
13:8	分值	该值的范围为 0～59
7:6	保留	
5:0	秒值	该值的范围为 0～59

2．完整时间寄存器 1 CTIME1

完整时间寄存器 1 包含的时间值为：日期（月）、月和年，如表 4-64 所列。

表 4-64　CTIME1 寄存器功能

位	功　能	说　明
31:28	保留	
27:16	年值	该值的范围为 0～4095
15:12	保留	
11:8	月值	该值的范围为 1～12
7:5	保留	
4:0	日期（月）值	该值的范围为 1～28、29、30 或 31

3．完整时间寄存器 2 CTIME2

完整时间寄存器 2 仅包含日期（年），如表 4-65 所列。

表 4-65　CTIME2 寄存器功能

位	功　能	说　明
31:12	保留	
11:0	日期（年）值	该值的范围为 1～365（闰年为 366）

时间计数器组包含 8 个可读写寄存器，用于 RTC 日历时间初始化，如表 4-66 所列。

表 4-66　时间计数器寄存器

名　称	规　格	说　明	访　问
SEC	6 位	初始化秒值，该值的范围为 0～59	R/W
MIN	6 位	初始化分值，该值的范围为 0～59	R/W
HOUR	5 位	初始化小时值，该值的范围为 0～23	R/W
DOM	5 位	初始化日期（月）值，该值的范围为 1～28、29、30 或 31	R/W
DOW	3 位	初始化星期值，该值的范围为 0～6	R/W
DOY	9 位	初始化日期（年）值，该值的范围为 1～365（闰年为 366）	R/W
MONTH	4 位	初始化月值，该值的范围为 1～12	R/W
YEAR	12 位	初始化年值，该值的范围为 0～4095	R/W

时间计数器的值在 RTC 运行过程中会发生变化，要人为改变 RTC 的值，必须通过重新设置时间计数器（SEC、MIN、HOUR、DOM、DOW、DOY、MONTH 和 YEAR）来实现。时间计数器的关系和值如表 4-67 所列。

表 4-67　时间计数器的关系和值

计 数 器	规　格	使　能	最 小 值	最 大 值
秒	6 位	Clk1	0	59
分	6 位	秒	0	59
小时	5 位	分	0	23
日期（月）	5 位	小时	1	28、29、30 或 31
星期	3 位	小时	0	6
日期（年）	9 位	小时	1	365 或 366(闰年)
月	4 位	日期（月）	1	12
年	12 位	月或日期（年）	0	4095

RTC 对于闰年的计算，是执行一个简单的位比较，看年计数器的最低两位是否为 0。如果为 0，那么 RTC 认为这一年为闰年。RTC 认为所有能被 4 整除的年份都为闰年。这个算法从 1901 年到 2099 年都是准确的，但在 2100 年出错，2100 年并不是闰年。闰年对 RTC 的影响只是改变 2 月份的天数、日期（月）和年的计数值。

报警寄存器组如表 4-68 所列。报警寄存器允许用户设定产生中断的日期或时间。这些寄存器的值与时间计数器相比较。如果未屏蔽的报警寄存器都与它们对应的时间计数器相匹配，则将产生一次中断。向中断位置寄存器 ILR 的位 1 写入 1 将清除中断。

表 4-68　报警寄存器

名　　称	规　　格	说　　明	访　　问
ALSEC	6 位	秒报警值	R/W
ALMIN	6 位	分报警	R/W
ALHOUR	5 位	小时报警	R/W
ALDOM	5 位	日期（月）报警	R/W
ALDOW	3 位	星期报警	R/W
ALDOY	9 位	日期（年）报警	R/W
ALMON	4 位	月报警	R/W
ALYEAR	12 位	年报警	R/W

预分频器允许从任何频率高于 65.536kHz 的外设时钟源产生一个 32.768kHz 的基准时钟。这样，不管外设时钟的频率为多少，RTC 总是以正确的速率运行。预分频器组包含一个整数寄存器和一个小数寄存器。

1．预分频整数寄存器 PREINT

预分频整数寄存器的功能如表 4-69 所列。

表 4-69　预分频整数寄存器功能

位	功　　能	说　　明	复　位　值
15:13	保留		NA
12:0	预分频整数值	包含 RTC 预分频值的整数部分	0

预分频值的整数部分计算公式如下：

$$PREINT = int（Fpclk/32768）-1$$

PREINT 的值必须大于等于 1。

2．预分频小数寄存器 PREFRAC

预分频小数寄存器的功能如表 4-70 所列。

表 4-70　预分频小数寄存器功能

位	功　　能	说　　明	复　位　值
15	保留		NA
14:0	预分频小数值	包含 RTC 预分频值的小数部分	0

预分频值的小数部分计算公式如下：

$$PREFRAC = Fpclk - ((PREINT +1)\times 32768)$$

4.10.3　应用举例

通过设置时钟控制寄存器（CCR）可以选择 RTC 的计数时钟源，可由独立的 32.769kHz 振荡器提供或由 Fpclk 的分频信号提供。当使用 Fpclk 作为时钟源时，它的基准时钟分频器允许调节任何频率高于 65.536kHz 的外设时钟源产生一个 32.768kHz 的基准时钟，实现准确的计时操作。

RTC 的寄存器框图如图 4-29 所示。实时时钟 CTC 计数器是一个 15 位的计数器，它位于秒计数器之前，CTC 每秒计数 32768 个时钟；当有 CTC 秒进位时，完整时间寄存器（CTME0～CTMIE2）、RTC 时间寄存器（SEC、MIN 等）将会更新；RTC 的中断有两种，一种是增量中断，由 CIIR 进行控制；另一种为报警中断，由 AMR 寄存器和各报警时间寄存器控制，如 ALSEC、ALMIN 等；报警位置寄存器 ILR 用来产生相应的中断标志；RTC 时钟控制寄存器 CCR 用于使能实时时钟，CTC 复位控制等。其中，用于表示“日”的日期寄存器有两个，分别为 DOY 和 DOM，DOY 表示为一年中的第几日，值为 1～365（闰年为 366）；而 DOM 则为一月中的第几日，值为 1～28/29/30/31，一般日期计数使用 DOM 即可。

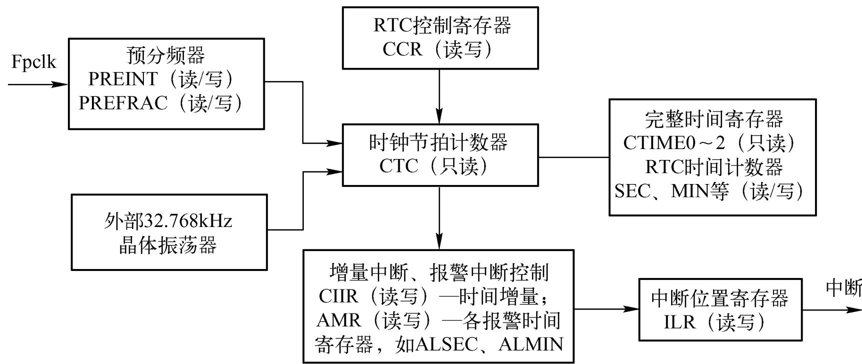


图 4-29　RTC 的寄存器功能框图

RTC 的基本操作方法如下：

- 选择时钟源（CCR 寄存器）。
- 设置 RTC 基准时钟分频器（PREINT、PREFRAC）（根据上一步选择执行此步操作）。
- 初始化 RTC 时钟值，如 YEAR、MONTH、DOM 等。
- 报警中断设置，如 CIIR、AMR 等。
- 启动 RTC，即 CCR 的 CLKEN 位置位。
- 读取完整时间寄存器值，或等待中断。

使用 RTC 时，VBAT 引脚必须连接到 V3 引脚，或者连到一个独立的电源（如外部电池），否则 VBAT 应该接地。VBAT 断电时，LPC2138 不能保存 RTC 的状态，如果时钟源丢失、中断或改变，RTC 也无法维持时间计数。RTC 可以使用两个时钟源（VPB 时钟 Fpclk 或来自 RTCX1～2 引脚的 32kHz 信号），如果将 RTCX1～2 引脚的 32kHz 信号作为时钟源，RTC 可以完全独立工作，与 VPB 时钟无关。因此，在要使用 RTC 且对功耗敏感的应用中（如电池供电设备），可通过使用 RTCX1～2 引脚的 32kHz 信号和清除 PCONP 功率控制寄存器中 PCRTC 位来降低功耗。

1．设置 RTC 时钟

```
PREINT = Fpclk / 32768 - 1;                    // 设置基准时钟分频器
PREFRAC = Fpclk - (Fpclk / 32768) * 32768;
```

2．设置 RTC 使用独立振荡器

```
CCR = 0x12;                                    // 使用独立振荡器
```

3．设置秒增量中断

```
CIIR = 0x01;                                    // 设置秒值的增量产生一次中断
```

4．设置 RTC 时间

```
CCR = 0x00;                                     // 禁止时间计数器
YEAR = 2005;
MONTH = 01;
DOM = 10;
DOW = 4;
HOUR = 8;
MIN = 30;
SEC = 59;
```

5．读取 RTC 时间

```
times = CTIME0;                                 // 读取完整的时钟寄存器
datas = CTIME1;
```

例 4−20 利用 RTC 实现万年历，通过 SPI 接口上的 74HC595 驱动 4 位 LED 数码管，能够显示出年、月、日、星期以及时间。利用按键中断控制显示状态，在年、月+日、时+分、秒+星期这 4 个状态之间反复切换，每按一次按键，显示状态切换一次。Proteus 仿真电路如图 4−30 所示。

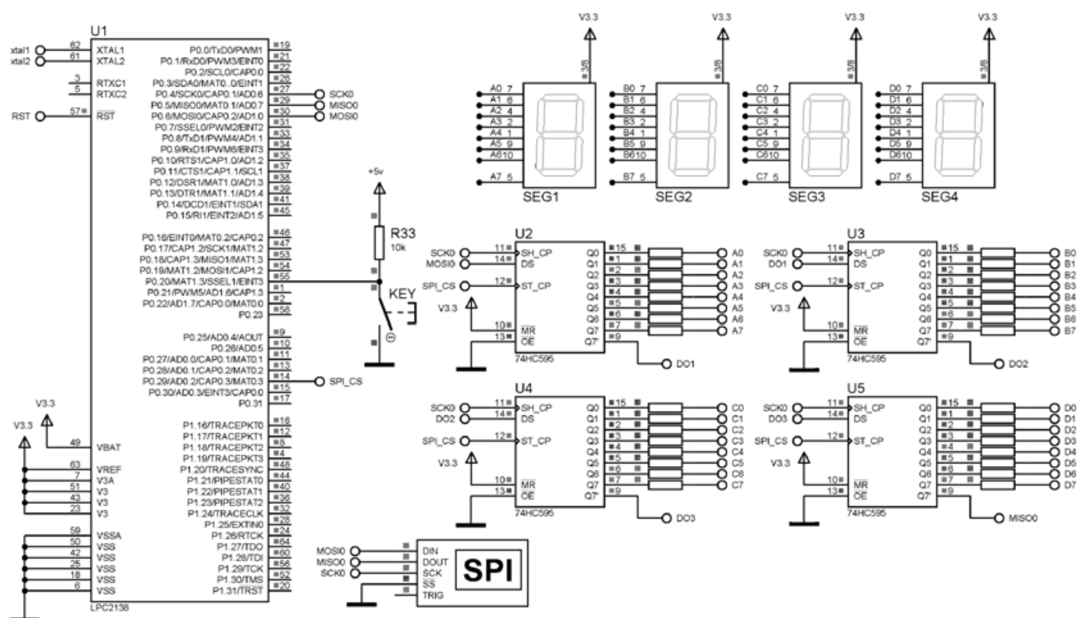


图 4−30 利用 RTC 实现万年历的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
```

```
#include    <intrinsics.h>

#define    HC595_CS        (1<<29)                /* P0.29 口为 74HC595 的片选 */
#define    KEY             (1<<20)

/****************************************************************************
* 函数名称：MSpiIni()
* 函数功能：初始化 SPI 接口，设置为主机
****************************************************************************/
void   MSpiIni(void)
{   VPBDIV = 0x02;
    S0SPCCR = 0x52;             // 设置 SPI 时钟分频
    S0SPCR = 0x30;              // 设置 SPI 接口模式，MSTR=1，CPOL=1，CPHA=0，LSBF=0
}

/****************************************************************************
* 函数名称：MSendData()
* 函数功能：向 SPI 总线发送数据。
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
****************************************************************************/
uint8   MSendData(uint8 data)
{
    uint8 temp = S0SPDR;
    IO0CLR = HC595_CS;                  // 片选
    IO0SET = HC595_CS;
    IO0CLR = HC595_CS;                  // 片选
    S0SPDR = data;
    while( 0==(S0SPSR&0x80) );          // 等待 SPIF 置位，即等待数据发送完毕
    IO0SET = HC595_CS;
    return(S0SPDR);
}

/* 此表为 0--F 的字模 */
uint8 const DISP_TAB[16] = { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,
0x88,0x83,0xC6,0xA1,0x86,0x8E};
/*******************************************************************************
** 函数名称：RTCInit()
** 函数功能：初始化实时时钟
*******************************************************************************/
void RTCInit (void)
{
      PREINT = Fpclk / 32768 - 1;                   // 设置基准时钟分频器
      PREFRAC = Fpclk - (Fpclk / 32768) * 32768;
      CCR     = 0x00;                               // 禁止时间计数器
      YEAR    = 2011;
      MONTH = 10;
```

```
    DOM     = 31;
    DOW     = 1;
    HOUR    = 8;
    MIN     = 30;
    SEC     = 59;
    CIIR = 0x01;                              // 设置秒值的增量产生 1 次中断
    CCR   = 0x01;                             // 启动 RTC
}

uint8   rcv_data,time_flag=1;

/****************************************************************************
* 函数名称：IRQ_Eint3()
* 函数功能：外部中断 EINT3 服务函数，改变时间显示标志
****************************************************************************/
void IRQ_Eint3(void)
{
   Delayms(10);                            //延时 10ms，防止按键断开时有干扰而误操作
   if((IO0PIN&KEY)==0)
   {
   time_flag++;
   if(time_flag==5)
   time_flag=1;
   }
   while((EXTINT&0x08)!=0)
   EXTINT = 0x08;                          // 清除 EINT3 中断标志
   VICVectAddr = 0x00;
}

/****************************************************************************
* 函数名称：main()
* 函数功能：万年历显示。利用中断按键控制显示状态，分别显示年、日期、时、分、
*           秒和星期 4 个状态, KEY 按键用来切换显示状态
****************************************************************************/
int   main(void)
{
   PINSEL0 = 0x00001500;                // 设置 SPI 引脚连接
   PINSEL1 = 0x00000301;                // 设置引脚连接，P0.20 设置为 EINT0
   IO0DIR = HC595_CS;                   // 设置 LED0 控制口为输出
   MSpiIni();                           // 初始化
   RTCInit();
   EXTMODE    = 0<<3;                   // 设置 EINT3 中断为边沿触发模式
   EXTPOLAR = 0<<0;                     // 设置 EINT3 中断为下降沿触发模式
   VICIntSelect = 0x00000000;           // 设置所有中断分配为 IRQ 中断
   VICVectCntl0 = 0x20|17;       // 0x20 表示向量 IRQ 使能，1<<17 表示 EINT3 在 VIC 通道 14 号
   VICVectAddr0 = (int)IRQ_Eint3;       // 设置中断服务程序地址
   EXTINT = 1<<3;                       // 清除 EINT3 中断标志
```

```
    VICIntEnable = (1<<17);                  // 使能 EINT3 中断
    __enable_irq();                          // 允许 IRQ 中断
    while(1)
    {
      switch(time_flag)
      {
       case 1:
       rcv_data = MSendData(DISP_TAB[YEAR % 10]);        //  发送年显示数据
       rcv_data = MSendData(DISP_TAB[YEAR%100/ 10]);
       rcv_data = MSendData(DISP_TAB[YEAR %1000/100]);
       rcv_data = MSendData(DISP_TAB[YEAR /1000]);
       Delayms(200);break;
       case 2:
       rcv_data = MSendData(DISP_TAB[DOM % 10]);         //  发送日期显示数据
       rcv_data = MSendData(DISP_TAB[DOM/ 10]);
       rcv_data = MSendData(DISP_TAB[MONTH %   10]);     //  发送月份显示数据
       rcv_data = MSendData(DISP_TAB[MONTH /10]);
       Delayms(200);break;
       case 3:
       rcv_data = MSendData(DISP_TAB[MIN % 10]);         //  发送分显示数据
       rcv_data = MSendData(DISP_TAB[MIN/ 10]);
       rcv_data = MSendData(DISP_TAB[HOUR %   10]);      //  发送时显示数据
       rcv_data = MSendData(DISP_TAB[HOUR /10]);
       Delayms(200);break;
       case 4:
       rcv_data = MSendData(DISP_TAB[DOW % 10]);         //  发送秒显示数据
       rcv_data = MSendData(DISP_TAB[DOW/ 10]);
       rcv_data = MSendData(DISP_TAB[SEC %   10]);       //  发送星期显示数据
       rcv_data = MSendData(DISP_TAB[SEC /10]);
       Delayms(200);break;
       default : break;
      }
    }
}
```

4.11 看门狗定时器

4.11.1 主要特性

LPC2138 具有 1 个看门狗定时器，其作用是使 CPU 在进入错误状态后的一定时间内复位。当看门狗使能时，如果用户程序没有在周期时间内喂狗（重装），看门狗会产生一个系统复位信号。

看门狗包括一个 4 分频的预分频器和一个 32 位计数器。时钟通过预分频器进入定时器，定时器递减计数。定时器递减的最小值为 0xFF，如果设置一个小于 0xFF 的值，系统会将 0xFF 装入计数器。因此最小看门狗间隔为(Tpclk×256×4)，最大间隔为(Tpclk×2^{32}×

4)，两者都是 Tpclk×4 的倍数。

看门狗的主要特性如下：

- 如果没有周期性重装，则产生片内复位。
- 调试模式。
- 由软件使能，但要求禁止硬件复位或看门狗复位/中断。
- 错误/不完整的喂狗时序会导致复位/中断（如果使能）。
- 指示看门狗复位的标志。
- 带内部预分频器的可编程 32 位定时器。
- 可选择 Tpclk×4 倍数的时间周期：从(Tpclk×256×4)到(Tpclk×2^{32}×4)。

4.11.2　寄存器描述

看门狗包含 4 个寄存器，分别介绍如下。

1．看门狗模式寄存器 WDMOD

WDMOD 寄存器通过 WDEN 和 RESET 这两位的组合来控制看门狗操作，其功能如表 4-71 所列。

表 4-71　WDMOD 寄存器功能

位	功　能	说　　明	复 位 值
0	WDEN	看门狗中断使能位（只能置位）	0
1	WDRESET	看门狗复位使能位（只能置位）	0
2	WDTOF	看门狗超时标志	0(外部复位)
3	WDINT	看门狗中断标志（只读）	0
7:4	保留		NA

WDMOD 寄存器通过 WDEN 和 RESET 的组合来控制看门狗的操作。如表 4-72 所列。

表 4-72　WDT 组和控制

WDEN	WDRESET	说　　明
0	X	看门狗关闭时的调试/操作
1	0	带看门狗中断的调试，但没有 WDRESET
1	1	带看门狗中断和 WDRESET 的操作

一旦 WDEN 和/或 WDRESE 位设置，就无法使用软件将其清零。这两个标志由外部复位或看门狗定时器溢出清零。

WDTOF：当看门狗发生超时，看门狗超时标志置位。该标志由软件清零。

WDINT：当看门狗发生超时，看门狗中断标志置位。产生的任何复位都会使该位清零。

2．看门狗定时器常数寄存器 WDTC

WDTC 是一个 32 位寄存器，它决定看门狗的超时值。当喂狗时序产生时，WDTC 的内容重新装入看门狗定时器。WDTC 寄存器的低 8 位的复位值为 1。写入一个小于 0xFF 的值时，会自动将 0xFF 值装入 WDTC，因此超时的最小时间间隔为 Tpclk×256×4。WDTC 寄存器的功能如表 4-73 所列。

表 4-73　WDTC 寄存器功能

位	功　能	说　明	复 位 值
31:0	计数值	看门狗超时间隔	0xFF

3．看门狗喂狗寄存器 WDFEED

先向该寄存器写入 0xAA，然后再写入 0x55，会使 WDTC 的值重新装入看门狗定时器。如果看门狗通过 WDMOD 寄存器使能，该操作还将启动看门狗运行。置位 WDMOD 中的 WDEN 位不足以使能看门狗。在看门狗能够产生中断/复位之前，必须完成一次有效的喂狗时序。否则，看门狗将忽略喂狗错误。

在一个喂狗时序中，一次对看门狗定时器寄存器不正确访问之后的第二个 Tpclk 周期将产生中断/复位。

如果应用程序开启了其他中断，为了保证喂狗操作的正确性，在喂狗之前要关闭中断，喂狗之后再开中断。

WDFEED 寄存器的功能如表 4-74 所列。

表 4-74　WDFEED 寄存器功能

位	功　能	说　明	复 位 值
7:0	喂狗	喂狗值应当先是 0xAA，然后是 0x55	未定义

4．看门狗定时器值寄存器 WDTV

WDTV 寄存器用于读取看门狗定时器的当前值，其功能如表 4-75 所列。

表 4-75　WDTV 寄存器功能

位	功能	说　明	复位值
31:0	计数	当前定时器值	0xFF

4.11.3　应用举例

看门狗的基本使用方法如下：

- 在 WDTC 寄存器中设置看门狗定时器的固定装载值。
- 在 WDMOD 寄存器中设置模式。
- 通过向 WDFEED 寄存器顺序写入 0xAA 和 0x55，启动看门狗。
- 在看门狗向下溢出之前应当再次喂狗以防止发生复位/中断。

当看门狗计数器向下溢出时，程序计数器将从 0x00000000 开始重新计数，和外部复位一样。可以通过检查看门狗超时标志（WDTOF），来确定看门狗是否产生复位条件。WDTOF 标志必须由软件清零。

1．设置看门狗模式

```
WDMOD = 0x03;                          // 设置看门狗模式：中断且复位
```

2．设置看门狗定时常数

```
WDTC = 0xff000;                        // 设置看门狗定时器参数
```

3．喂狗操作

```
WDFEED = 0xAA;                    // 注意：第一次喂狗启动看门狗
WDFEED = 0x55;
```

4．喂狗之前关中断

```
IRQDisable(); 或者 FIQDisable();       // 喂狗之前关中断
WDFEED = 0xAA;
WDFEED = 0x55;
IRQEnable(); 或者 FIQEnable();         // 喂狗之后开中断
```

例 4-21 利用看门狗溢出复位。程序设定看门狗溢出产生复位，正常情况下，LED1 闪烁，并周期性喂狗，防止看门狗溢出。如果 KEY1 键长时间按下时喂狗周期被打断，超过 WDTC 设定时间，看门狗将会溢出并产生复位。发生 WDT 复位后，程序判断看门狗溢出标志 WDTOF 是否已经被清除，若没有被清除，则 LED2 闪烁报警，并等待软件清除 WDTOF；若在此期间 KEY2 键被按下，则清除 WDTOF，程序又回到正常情况。Proteus 仿真电路如图 4-31 所示。

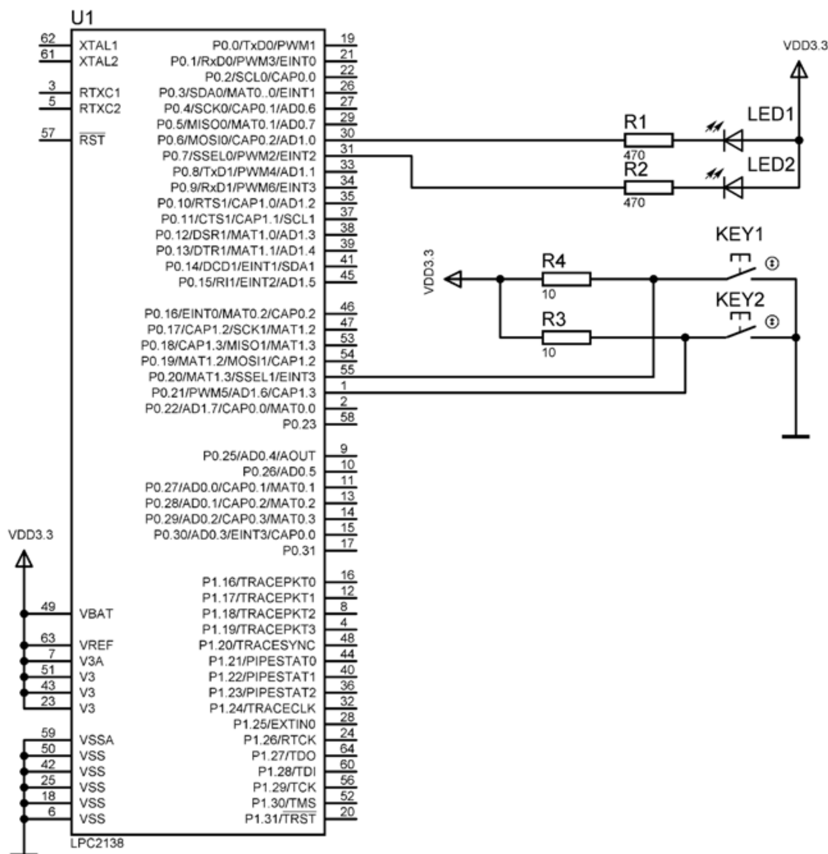


图 4-31　看门狗溢出产生复位的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"

#define     LED1       1 << 6          // P0.6 控制 LED1
#define     KEY1       1 << 20         // P0.20
#define     KEY2       1 << 21         // P0.21
#define     LED2       1 << 7          // P0.7 控制 LED2

/****************************************************************************************
* 函数名称：main()
* 函数功能：正常情况下 LED1 闪烁，按下按键 KEY1 停止喂狗，看门狗超时将主程序复位，LED2
           闪烁，此时如果按下按键 KEY2，清除看门狗溢出标志位，程序正常运行，LED1 闪烁
****************************************************************************************/
int   main(void)
{     uint32 i,j;
      PINSEL2= 0X00;                          // 引脚连接
      IO0DIR = 0XFF;                          // 控制口输出
      IO0CLR = 0XFF;
      IO0SET = LED2;                          // 关闭 LED2

       /* 如果曾经发生过看门狗复位，则等待清除看门狗超时标志，在未清除之前 LED2 一直闪烁 */
      while(( WDMOD & 0x04 ) == 0x04)         // 判断看门狗超时标志
      {        /* 看门狗超时标志未清除 */
            for(j = 0; j < 0xFFFF; j++);      // LED2 闪烁周期
            if((IO0SET & LED2) == 0)
            {
                  IO0SET = LED2;              // 关闭 LED2
                  if((IO0PIN & KEY2) == 0)    // 等待按键 KEY2
                  WDMOD = 0x00;               // 如果 KEY2 按下，则软件清零看门狗超时标志位
            }
            else      IO0CLR = LED2;          // LED2 闪烁
      }

      /* 如果没有发生看门狗复位或者看门狗超时标志已经清除，则初始化看门狗 */
     WDTC    = 0xff000;                       // 设置看门狗定时器参数
     WDMOD = 0x03;                            // 设置看门狗模式：中断且复位
      WDFEED = 0xAA;                          // 第一次喂狗启动 WDT
     WDFEED = 0x55;

      /* 正常情况下，LED1 闪烁 */
      while(1)
      { /* 用按键 KEY1 模拟导致无法周期性喂狗的意外情况 */
           while((IO0PIN & KEY1) == 0)        // 如果按键 KEY1 按下，就停止喂狗
           {for(i = 0; i < 0x0FFFF; i++); }
           for(j = 0; j < 0xFF; j++)          // LED1 闪烁周期
           {
```

```
                for(i = 0; i < 0x0FFF; i++);            // 喂狗周期
                WDFEED = 0xAA;                          // 喂狗时序
                WDFEED = 0x55;
            }
            /* LED1 闪烁  */
            if((IO0SET & LED1) == 0)    IO0SET = LED1;
            else      IO0CLR = LED1;
        }
    }
```

例 4−22 看门狗使能后，需要在规定周期内重新喂狗，否则会产生看门狗溢出。看门狗溢出可产生中断或者复位，本例利用看门狗溢出产生中断。确定看门狗溢出时间值，并且设定 WDMOD 为溢出产生中断。程序正常运行的时候，LED1 闪烁，如果长时间按下 KEY1 键，喂狗周期将被人为延长，当时间超过 WDTC 所设定的周期，将触发 WDT 溢出中断，点亮 LED2 报警。看门狗一旦出现溢出，其产生的中断标志位 WDINT 将不能被软件清除，而只能通过复位清除，因此看门狗产生中断后就不能再次产生中断，除非重新复位芯片。Proteus 仿真电路如图 4−32 所示。

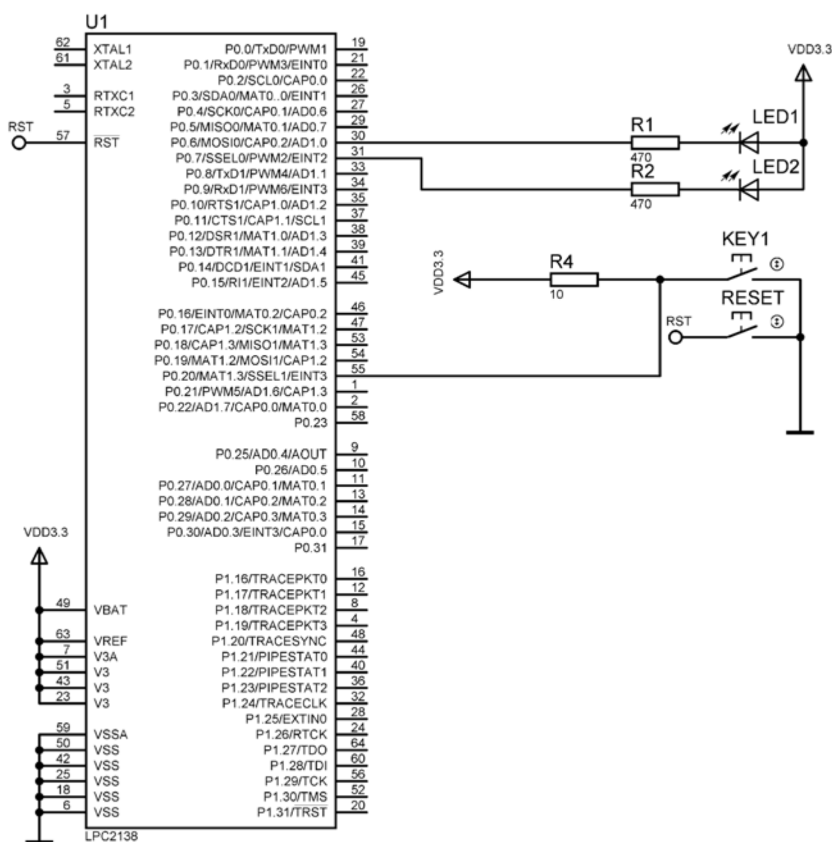


图 4−32　看门狗溢出产生中断的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>

#define     LED2      (1 << 7)              // P0.7 控制 LED2，低电平点亮
#define     LED1      (1 << 6)              // P0.6 控制 LED1，低电平点亮
#define     KEY1      (1 << 20)             // P0.20 连接 KEY1
#define     WDT       0                     // WDT 中断号

/*********************************************************************************
** 函数名称：WDT_Int()
** 函数功能：WDT 中断服务函数
*********************************************************************************/
void WDT_Int(void)
{
     IO0CLR   = LED2;                       // LED2 点亮
     WDMOD    = 0x00;                       // 清除看门狗超时标志 WDTOF
     VICIntEnClr = 1 << WDT;                // 看门狗溢出中断只能通过禁止 VIC 的方式返回
     VICVectAddr = 0x00;                    // 通知 VIC 中断结束
}

/**************************************************************************
* 函数名称：main()
* 函数功能：循环喂狗，如果按下 KEY1 键就停止喂狗，看门狗溢出产生中断
**************************************************************************/
int   main(void)
{    uint32 i, j;
     PINSEL0=0x00;
     IO0DIR =0xFF ;
     IO0SET = LED2;
     IO0SET = LED1;
       /* 中断初始化 */
     VICIntSelect = 0x00000000;             // 连接所有中断到向量中断
     VICVectCntl0 = 0x20 | WDT;             // 分配 WDT 中断到 slot0
     VICVectAddr0 = (int)WDT_Int;           // 中断服务程序地址
     WDMOD = 0x00;                          // 清除看门狗超时位 WDTOF
     VICIntEnable = (1 << WDT);             // 中断使能

     /* 初始化看门狗 */
     WDTC   = 0x6006;                       // 设定看门狗超时值
     WDMOD = 0x01;                          // 看门狗中断使能
     WDFEED = 0xAA;                         // 第一次喂狗启动 WDT
     WDFEED = 0x55;
     __enable_irq();                        // 开中断
```

```
while (1)
{       /* 周期性喂狗，如果 KEY1 按下，就停止喂狗 */
        while((IO0PIN & KEY1) == 0);
        for(j = 0; j < 0xff; j++)
        {     /* 循环喂狗 */
                for(i = 0; i < 0x0FFF; i++);
                WDFEED = 0xAA;
                WDFEED = 0x55;
        }
      /* 闪烁 LED1 */
      if((IO0SET & LED1) == 0) IO0SET = LED1;
                else IO0CLR = LED1;
        }
}
```

复习思考题

1．使用 GPIO 输出功能设计一个花样流水灯。

2. 使用 GPIO 输入功能检测按键 KEY1～KEY8 的状态，使用 GPIO 输出功能驱动发光二极管 LED1～LED8，要求当按键按下时点亮 LED，按键松开时 LED 熄灭。

3．简述 LPC2138 UART 的基本操作方法。

4．分别采用查询和中断方式设计 LPC2138 与 PC 通信，通过 UART0 接收 PC 发送的字符串，然后送回 PC 机显示。

5．LPC2138 的硬件 I^2C 接口具有哪些主要特性？

6．简述 LPC2138 的硬件 I^2C 接口主机基本操作方法。

7．简述 LPC2138 的硬件 I^2C 接口从机基本操作方法。

8．利用 LPC2138 硬件 I^2C 接口，对串行 EEPROM 器件 24C02 进行读/写，绘制原理电路并编制相应程序。

9．LPC2138 的硬件 SPI 接口具有哪些主要特性？

10. 利用 LPC2138 的 SPI 接口作为主机，对串行存储器 25C160 进行读/写, 绘制原理电路并编制相应程序。

11．LPC2138 的硬件 SSP 接口具有哪些主要特性？

12. 利用 LPC2138 的 SPI 接口作为主机，控制 74HC595 驱动 LED 数码管显示器，绘制原理电路并编制相应程序。

13．LPC2138 的定时器/计数器具有哪些主要特性？

14．简述 LPC2138 的定时器/计数器的基本操作方法。

15．设计利用 LPC2138 定时器 T0 控制发光二极管 LED，以查询方式实现 1s 定时，让 LED 每 1s 点亮一次。绘制原理电路并编制相应程序。

16．利用 LPC2138 定时器 T0 的计数功能，对按键压下所产生的脉冲信号进行计数，并通过数码管显示脉冲计数值。绘制原理电路并编制相应程序。

17．LPC2138 的硬件 PWM 具有哪些主要特性？

18．简述 LPC2138 的硬件 PWM 的基本操作方法。

19．利用 PWM 输出占空比可调节的方波波形，每按一次按键 KEY，输出波形的占空比发生一次改变。绘制原理电路并编制相应程序。

20．LPC2138 片内 A-D 转换器具有哪些主要特性？

21．设置 P0.27 连接 AD0.0，P0.28 连接 AD0.1，对外部输入电压采样，进行 A-D 转换，转换后的电压值通过 UART1 输出，通过虚拟终端观察转换结果。绘制原理电路并编制相应程序。

22．LPC2138 片内 D-A 转换器具有哪些主要特性？

23．利用 LPC2138 片内 D-A 转换器，产生三角波、锯齿波、正弦波，通过按键进行输出波形切换。绘制原理电路并编制相应程序。

24．LPC2138 片内实时时钟 RTC 具有哪些主要特性？

25．简述 LPC2138 片内实时时钟 RTC 的基本操作方法。

26．LPC2138 看门狗定时器具有哪些主要特性？

27．简述 LPC2138 看门狗定时器的基本操作方法。

第 5 章　LPC2138 片外扩展功能应用技术

5.1　液晶显示器 LCD 接口技术

对于采用电池供电的便携式应用系统，考虑到其低功耗的要求，常常需要采用液晶显示器 LCD（Liquid Crystal Diodes）。液晶显示器 LCD 体积小，重量轻，功耗极低，因此在仪器仪表中的应用十分广泛。LCD 是一种被动式显示器，它本身并不发光，只是调节光的亮度。目前常用的 LCD 是根据液晶的扭曲——向列效应原理制成的。这是一种电场效应，夹在两块导电玻璃电极之间的液晶经过一定处理后，其内部的分子呈 90°的扭曲，这种液晶具有旋光特性。当线性偏振光通过液晶层时，偏振面会旋转 90°。当给玻璃电极加上电压后，在电场的作用下液晶的扭曲结构消失，其旋光作用也随之消失，偏振光便可以直接通过。当去掉电场后液晶分子又恢复其扭曲结构。把这样的液晶放在两个偏振片之间，改变偏振片的相对位置（平行或正交）就可得到黑底白字或白底黑字的显示形式。

5.1.1　点阵字符型液晶显示模块接口技术

点阵字符型液晶显示模块能显示数字、字母、符号以及少量自定义图形符号，如简单汉字等，因而得到了广泛应用。点阵字符型液晶显示模块由液晶显示器、点阵驱动器、LCD 控制器等组成，模块内集成有字符发生器和数据存储器，采用单一+5V 电源供电。点阵字符型液晶显示模块在国际上已经规范化，所采用的控制器多为日立公司的 HD44780 及其兼容电路，如 SED1278（SEIKO、EPSON 公司产品）、KS0066（三星公司产品）等。表 5-1 列出了 EPSON 公司生产的 EA-D 系列各型号点阵字符型液晶显示模块的外部特性。

表 5-1　EA-D 系列点阵式液晶显示模块外部特性

名　称	字符数（列×行）	外部尺寸/mm	视觉范围/mm	字符点阵	字符尺寸/mm	点的尺寸/mm	速　率
EA-D16015	16×1	80×36	64.5×13.8	5×7	3.07×6.56	0.55×0.75	1/16
EA-D16025	16×2	84×44	61.×15.8	5×7	2.96×5.56	0.56×0.66	1/16
EA-D20025	20×2	116×37	83.×18.6	5×7	3.20×5.55	0.60×0.65	1/16
EA-D20040	20×4	98×60	76.×25.2	5×7	3.01×4.84	0.57×0.57	1/16
EA-D24016	24×1	126×36	100.0×13.8	5×10	3.15×8.70	0.55×0.70	1/11
EA-D40016	40×1	182×33.5	154.4×15.8	5×10	3.15×8.70	0.55×0.70	1/11
EA-D40025	40×2	182×33.5	154.4×15.8	5×7	3.20×5.55	0.60×0.65	1/16

下面介绍 EPSON 公司生产的点阵字符型液晶显示模块的接口及应用。EA-D 系列点阵字符型液晶显示模块内部结构如图 5-1 所示。它由点阵式液晶显示面板、SED1287 控制器和 4 个列驱动器组成。SED1278 完成显示模块的时序控制，同时也可以驱动 16 行 40 列的点阵库。

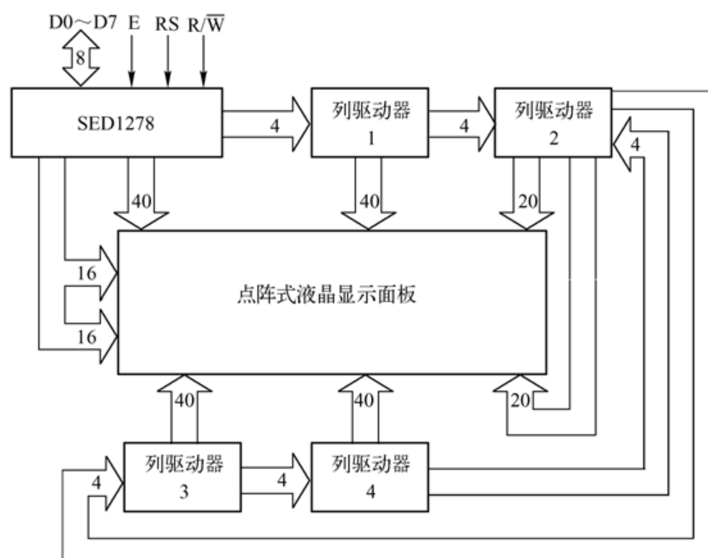


图 5-1　EPSON 公司的点阵字符型液晶显示模块内部结构

SED1278 控制器有 14 条引脚：

- VSS：地线输入端。
- VDD：+5V 电源输入端。
- VO：液晶显示面板亮度调节，通过 10～20kΩ的电阻接到+5V 和地之间，起调节显示亮度的作用。
- RS：寄存器选择信号输入线，低电平选通指令寄存器，高电平选通数据寄存器。
- R/$\overline{W}$：读/写信号输入线，低电平为写入，高电平为读出。
- E：片选信号输入线，高电平有效。
- D0～D7：数据总线，可以选择 4 位总线或 8 位总线操作，选择 4 位总线操作时使用 D4～D7。

SED1287 的控制电路主要由指令寄存器 IR、数据寄存器 DR、忙标志 BF、地址计数器 AC、预定义字符发生器 CGROM、自定义字符发生器 CGRAM、显示数据寄存器 DDRAM 以及时序发生电路所组成。

指令寄存器 IR 用于寄存各种指令码，只能写入，不能读出。

数据寄存器 DR 用于寄存显示数据，由内部操作自动写入 DDRAM 和 CGRAM，或寄存从 DDRAM 和 CGRAM 读出的数据。

忙标志 BF=1 时，表示正在进行内部操作，此时不能接受任何外部指令和数据。

地址计数器 AC 作为 DDRAM 或 CGRAM 的地址指针。如果地址码随指令写入 IR，则 IR 的地址码自动装入 AC，同时选择 DDRAM 或 CGRAM 单元。

显示数据寄存器 DDRAM 用于存储显示数据，DDRAM 的地址与显示屏幕的物理位置是一一对应的，当向显示数据寄存器某一地址单元写入一个字符的编码时，该字符就在对应的位置上显示出来。表 5-2 列出了 DDRAM 显示地址与显示屏物理位置的对应关系。

表 5-2　DDRAM 显示地址与显示屏物理位置关系

显示地址 列号 / 行号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	51	52	53
3	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	20	21	22	23	24	25	26	27
4	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67

预定义字符发生器 CGROM 由 8 位字符码生成 5×7 点阵字符 160 个和 5×10 点阵字符 32 个，已经固化在液晶显示器模块内部，由用户随意使用。表 5-3 列出了 8 位字符编码的高、低位排列及其与字符的对应关系，如果想显示 192 个字符中的一个，只要把该字符的编码送入 DDRAM 即可，如果想显示 192 个字符以外的字符，则需要利用自定义字符发生器 CGRAM 来定义特殊字符。

表 5-3　CGROM 字符编码表

高位 / 低位	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
0000		0	@	P	\	p		-	ﾀ	ﾐ	α	p
0001	!	1	A	Q	a	q	•	ヌ	チ	ム	a	q
0010	“	2	B	R	b	r	「	イ	ツ	メ	β	θ
0011	#	3	C	S	c	s	」	ウ	チ	モ	ε	∞
0100	$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
0101	%	5	E	U	e	u	。	オ	ナ	ユ	σ	O
0110	&	6	F	V	f	v	ラ	カ	ニ	ヨ	ρ	Σ
0111	，	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
1000	（	8	H	X	h	x	ィ	ク	ネ	リ	∫	X
1001	）	9	I	Y	i	y	ゥ	ケ	ノ	ル	−1	Y
1010	*	：	J	Z	j	z	ェ	コ	ハ	レ	j	千
1011	+	；	K	[	k	{	ォ	サ	ヒ	ロ	×	万
1100	，	<	L	¥	l	|	セ	シ	フ	ワ	Φ	⊕
1101	−	=	M	]	m	}	コ	ス	ヘ	ン	£	÷
1110	.	>	N	∧	n	→	ョ	セ	ホ	ハ	n	
1111	/	?	O	−	o	←	ッ	ソ	マ	ロ	○	■

自定义字符发生器 CGRAM 是为用户创建自己的特殊字符而设立的，它的容量仅为 64B，地址为 00H～3FH，但是作为自定义字符字模使用的仅是一个字节中的低 5 位，每个字节的高 3 位可作为数据存储器使用。若自定义字符为 5×7 点阵，可定义 8 个字符，自定义字符的编码为 00H～07H。

表 5-4 列出了自定义字符“上”，从表中可以看出，字符编码（DDRAM 中的数据）的第 0～2 位等同于 CGRAM 地址的第 3～5 位。CGRAM 地址的第 0～2 位定义字符数据的行

位置。CGRAM 中数据的 0～4 位决定字符形式，第 4 位是字符的最左端。CGRAM 的第 5～7 位不用作显示字符，因此它可用作一般的数据 RAM。

表 5-4　CGRAM 自定义字符

字符编码（DDRAM 数据）	CGRAM 地址	字符形式（CGRAM 数据）							
76543210	543210	7	6	5	4	3	2	1	0
	000	×	×	×	0	0	1	0	0
	001	×	×	×	0	0	1	0	0
	010	×	×	×	0	0	1	0	0
0000 ×000	000011	×	×	×	0	0	1	1	1
	100	×	×	×	0	0	1	0	0
	101	×	×	×	0	0	1	0	0
	110	×	×	×	1	1	1	1	1
	111	×	×	×	0	0	0	0	0

点阵字符型液晶显示模块的显示功能是由各种命令来实现的，共有 11 条命令。

1．清显示命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1

该命令把空格编码 20H 写入显示数据存储器的所有单元。

2．光标返回命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	1	×

该命令把地址计数器中 DDRAM 地址清 0，如果显示屏上显示了字符，则光标移到起始位置。如果显示两行，则光标移到第一行第一个字符的位置，显示数据存储器的内容不变。

3．设置输入方式命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1	I/D	S

当一个字符编码被写入 DDRAM 或从 DDRAM 中读出时，若 I/D=1，则 DDRAM 地址加一，若 I/D=0 则 DDRAM 地址减一。地址加一时，光标右移；地址减一时，光标左移。对 CGRAM 的读写操作和 DDRAM 一样，只是 CGRAM 与光标无关。当 S=1 时，整个显示屏向左（I/D=1）或向右（I/D=0）移动。在从 DDRAM 中读数、向 CGRAM 写数或从 CGRAM 中读数、S=0 这三种情况下，显示屏不移动。

4．显示开/关控制命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	1	D	C	B

当 D＝0 时，显示器关闭，显示数据存储器中的数据不变；当 D＝1 时，显示器立即显示 DDRAM 中的数据。

当 C＝0 时，不显示光标；当 C=1 时，显示光标。当选择字符为 5×7 点阵时，用第 8 行的第 5 个点显示光标。

当 B=1 时，显示闪烁光标，当时钟为 270kHz 时，在 379.2ms 内交换显示全黑点和字符，以实现字符闪烁。

5．光标或显示屏移动命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	S/C	R/L	×	×

该命令使显示和光标向左或向右移位。对两行显示而言，光标从第一行的第 40 个字符位置移到第二行的首位。从第二行的第 40 个位置不能移位到清屏的起始位置，而是回到第二行的第一个位置。命令中 S/C 和 R/L 位的作用如表 5-5 所示。

表 5-5　S/C 和 R/L 的作用

S/C	R/L	作　用
0	0	光标左移，地址计数器减一
0	1	光标右移，地址计数器加一
1	0	显示屏左移，光标跟随显示屏移动
1	1	显示屏右移，光标跟随显示屏移动

6．功能设置命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	IF	N	F	×	×

命令中的 IF 位用来设置接口数据长度，当 IF=1 时，数据以 8 位长度（D7～D0）发送或接收，当 IF=0 时，数据以 4 位长度（D7～D4）发送或接收。命令中的 N 和 F 位用来设置显示屏的行数和字符的点阵。设置方式如表 5-6 所示。

表 5-6　控制位 N、F 的设置

N	F	显示行数	字符点阵	占空系数
0	0	1	5×7	1/16
0	1	1	5×10	1/11
1	0	2	5×7	1/16
1	1	2	5×7	1/16

对于 EA-D20040 来说一定要设置 N=1，显示两行。

7．设置 CGRAM 地址命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	A5	A4	A3	A2	A1	A0

该命令的功能是设置 CGRAM 的地址，命令执行后，系统和 CGRAM 可连续进行数据交换。

8．设置 DDRAM 地址命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	A6	A5	A4	A3	A2	A1	A0

该命令的功能是设置 DDRAM 的地址，命令执行后，系统与 DDRAM 进行数据交换。

9．读忙标志和地址命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	1	BF	A6	A5	A4	A3	A2	A1	A0

该命令的功能是读出忙标志 BF 的值。当读出的 BF=1 时，则说明系统内部正在进行操作，不能接收下一条命令。在读出 BF 值的同时，CGRAM 和 DDRAM 所使用的地址计数器的值也被同时读出。

10．向 CGRAM 或 DDRAM 写数据命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	0	D	D	D	D	D	D	D	D

该命令的功能是把二进制数 DDDDDDDD 写入 CGRAM 或 DDRAM 中，若先送入 CGRAM 的地址则向 CGRAM 写入；若先送入 DDRAM 地址则向 DDRAM 写入。

11．从 CGRAM 或 DDRAM 读取数据命令

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
1	1	D	D	D	D	D	D	D	D

该命令的功能是将数据从用写数据命令建立的 CGRAM 或 DDRAM 地址指出的 RAM 中读出。在本命令之前的命令应是 CGRAM 或 DDRAM 地址建立命令、光标移位命令、或是上次 CGRAM/DDRAM 数据读出命令，若是其他命令，读出的数据可能会出错。

在执行读数据或写数据命令之后，地址计数器会自动加一或减一。一般是先执行一条地址建立命令或光标移位命令，再执行读数据命令，一旦一条读数据命令被执行后，就可连续执行数据读取命令，而不需再执行其他命令了。

例 5-1　图 5-2 所示为 LPC2138 与点阵字符型液晶显示模块的一种接口电路。液晶显示模块的 R/W、RS 和 E 信号分别由 LPC2138 的 P0.5、P0.6 和 P0.7 控制，P0.8～P0.15 用作显

示模块的 8 位数据线，这种方式称为间接式接口。

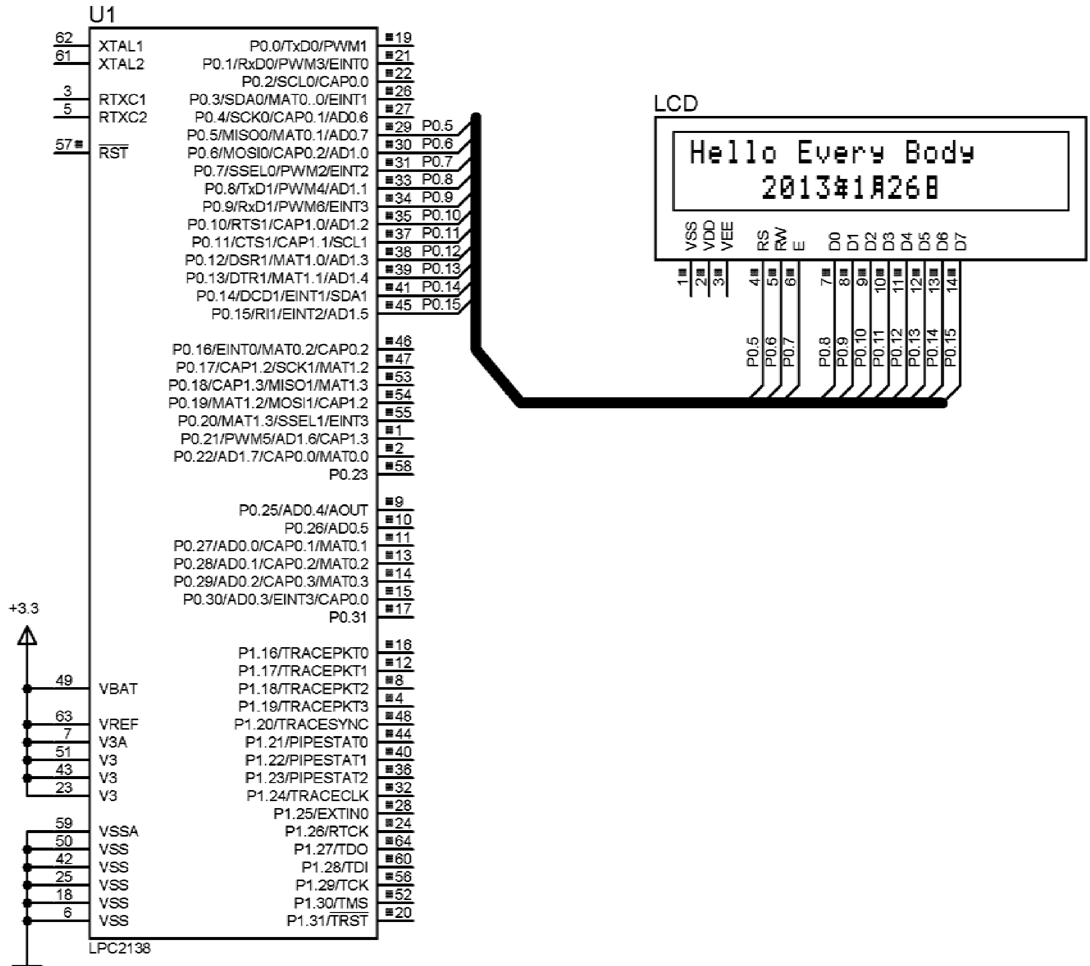


图 5-2　LPC2138 与点阵字符型液晶显示模块的接口电路

进入主程序后首先调用液晶模块初始化子程序，初始化内容包括将功能设置（8 位字长、2 行、5×7 点阵）、清屏、设置输入方式和设置显示方式及光标等，需要注意的是，每写入一条命令，都应先检查忙标志 BF，只有当 BF=0 时，才能执行下一条指令。接着调用自定义汉字字符子程序，该子程序中先设定 CGRAM 首地址，然后依次向 CGRAM 中写入各个自定义汉字的字模数据；接着设定显示字符在液晶屏上的位置，即 DDRAM 的地址，最后将要显示的字符代码分别写入 DDRAM，对于 CGROM 中的字符代码，可以直接以 ASCII 码表示（双引号中的字符串编译后自动生成 ASCII 码），而自定义汉字字符的代码则为 00H～07H，本例只定义了 3 个字符“年”、“月”、“日”，它们的代码分别为 00H、01H 和 02H。

主程序文件 main.c 如下：

```
#include    "config.h"
```

```
#define rs (1<<5)
#define rw (1<<6)
#define en (1<<7)
#define busy (1<<15)

/**********************************************************************************
* 函数名称：ChkBusy()
* 函数功能：检查总线是否忙
**********************************************************************************/
void ChkBusy(){
	IO0DIR=0xf0;
	while(1){
		IO0CLR=rs;
		IO0SET=rw;
		IO0SET=en;
		if(!(IO0PIN & busy))break;
		IO0CLR=en;
	}
	IO0DIR=0xffffffff;
}

/**********************************************************************************
* 函数名称：WrOp()
* 函数功能：写命令
* 入口参数：dat，待写入 LCD 的命令
**********************************************************************************/
void WrOp(uint8 dat){
	ChkBusy();
	IO0CLR=rs;
	IO0CLR=rw;
	IO0CLR=IO0CLR|0xff00;		//先清零
	IO0SET=dat<<8;			//再送数
	IO0SET=en;
	IO0CLR=en;
}

/**********************************************************************************
* 函数名称：WrDat()
* 函数功能：写数据
* 入口参数：dat，待写入 LCD 的数据
**********************************************************************************/
void WrDat(uint8 dat){
	ChkBusy();
	IO0SET=rs;
	IO0CLR=rw;
```

```
	IO0CLR=IO0CLR|0xff00;		//先清零
	IO0SET=dat<<8;			//再送数
	IO0SET=en;
	IO0CLR=en;
}

/*************************************************************************************
* 函数名称：lcd_init()
* 函数功能：lcd 初始化
*************************************************************************************/
void lcd_init(void){
	WrOp(0x38);
	WrOp(0x06);
	WrOp(0x0c);
}

/*************************************************************************************
* 函数名称：LocateXY()
* 函数功能：显示光标定位
* 入口参数：posx LCD 列位置，posy LCD 行位置
*************************************************************************************/
void LocateXY( uint8 posx,uint8 posy) {
	uint8 temp;
	temp = posx & 0xf;
	posy &= 0x1;
	if ( posy )temp |= 0x40;
	temp |= 0x80;
	WrOp(temp);
}

/*************************************************************************************
* 函数名称：DispOneChar()
* 函数功能：显示单字符
* 入口参数：x LCD 列位置，y LCD 行位置，Wdata  显示字符
*************************************************************************************/
void DispOneChar(uint8 x,uint8 y,uint8 Wdata) {
	LocateXY( x, y );		// 定位显示字符的 x、y 位置
	WrDat( Wdata );			// 写字符
}

/*************************************************************************************
* 函数名称：ePutstr()
* 函数功能：显示字符串
* 入口参数：x LCD 列，y LCD 行，j  字符串长度，*ptr  字符串指针
*************************************************************************************/
void ePutstr(uint8 x,uint8 y, uint8 j,uint8 *ptr){
```

```
        uint8 i,l=0;
        for (i=0;i<j;i++){
                DispOneChar(x++,y,ptr[i]);
                if ( x == 16 ){
                        x = 0; y ^= l;
                }
        }
}

/***********************************************************************************
* 函数名称：DisText()
* 函数功能：显示文本
* 入口参数：addr    LCD DDRAM 地址，*p    文本指针
***********************************************************************************/
void DisText(uint8 addr,char *p){
        WrOp(addr);
        while(*p !='\0')WrDat(*(p++));
}

/***********************************************************************************
* 函数名称：main()
* 函数功能：LCD 初始化，自定义汉字字符，LCD 显示英文和汉字字符
***********************************************************************************/
int    main(void){
        uint8 i;
        uint8 examp1[]={0x32,0x30,0x31,0x33,0x00,0x31,0x01,0x32,0x36,0x02};
        uint8 Hzzimo[]={0x08,0x0F,0x12,0x0F,0x0A,0x1F,0x02,0x00,         // 年
                        0x0F,0x09,0x0F,0x09,0x0F,0x09,0x11,0x00,         // 月
                        0x0F,0x09,0x09,0x0F,0x09,0x09,0x0F,0x00};        // 日
        lcd_init();                                 // LCD 初始化
        WrOp( 0x40 );                               // 自定义汉字字符：年、月、日
        for (i=0;i<24;i++){
                WrDat(Hzzimo[i]);
        }
        DisText(0x80,"Hello Every Body");           // 第一行从第 0 位开始显示英文字符
        ePutstr(4,1,10,examp1);                     // 第二行从第 4 位开始显示 2013 年 1 月 26 日
        while(1);
}
```

5.1.2　12864 点阵图型液晶显示模块接口技术

点阵字符型液晶显示模块只能显示英文字符和简单的汉字，要想显示较为复杂的汉字或图形，就必须采用点阵图型液晶显示模块，本节介绍 12864 点阵图型液晶显示模块与 LPC2138 的接口技术。12864 液晶显示模块内部控制器采用 KS0108 或 HD61202，图 5-3 所示为其引脚排列，引脚功能如表 5-7 所列。

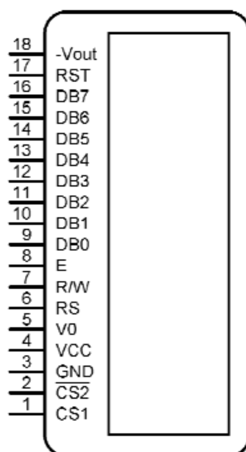


图 5-3 12864 点阵图型液晶显示模块引脚排列

表 5-7 12864 点阵图型液晶显示模块的引脚功能

引脚	符号	功能	引脚	符号	功能
1	$\overline{CS1}$	1：选择左边 64×46 点	7	R/W	1：数据读取，0：数据写入
2	$\overline{CS2}$	1：选择右边 64×46 点	8	E	使能信号，负跳变有效
3	GND	地	9～16	DB0～DB7	数据信号
4	VCC	+5V 电源	17	$\overline{RST}$	复位，低电平有效
5	VO	显示驱动电源 0～5V	18	-Vout	LCD 驱动负电源
6	RS	1：数据输入，0：命令输入			

12864 内部存储器 DDRAM 与显示屏上的显示内容具有一一对应关系，用户只要将显示内容写入到 12864 内部显示存储器 DDRAM 中，就能实现正确显示。12864 液晶屏横向有 128 个点，纵向有 64 个点，分为左半屏和右半屏，DDRAM 与显示屏的对应关系如表 5-8 所列。

表 5-8 12864 内部 DDRAM 液晶显示屏的关系

$\overline{CS1}$ =1(左半屏)						$\overline{CS2}$ =1(右半屏)					
Y=	0	1	……	62	63	0	1	……	62	63	行号
X=0	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	0 ↓ 7
X=1	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	8 ↓ 15
……	……	……	……	……	……	……	……	……	……	……	……
X=7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	DB0 ↓ DB7	56 ↓ 63

在 12864 液晶屏上显示图形或汉字时，可以利用字模提取软件获得图形或汉字的点阵代码。以“嵌”字 16×16 点阵显示为例，按纵向取模方式获得的字模点阵数据如下：

```
0x40,0x40,0xFE,0x48,0x48,0x48,0xF8,0x4F,
0x08,0xC8,0x78,0x48,0x5E,0xC0,0x00,0x00,
0x00,0x00,0xFF,0x44,0x44,0x44,0x7F,0x80,
```

0x41,0x30,0x0F,0x10,0x61,0xC0,0x40,0x00,

字模点阵数据是纵向的，一个像素对应一个位。8 个像素对应一个字节，字节的位顺序是上低下高。例如，从上到下 8 个点的状态是“*-----*-” (*为黑点，-为白点)，则转换的字模数据是 41H(01000001B)。显示时先输入汉字的上半部分 16 个数据，再输入下半部分 16 个数据。

12864 点阵图型液晶显示模块的指令功能比较简单, 共有 8 条指令。

1．读忙标志

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	1	1	BUSY	0	ON/OFF	RESET	0	0	0	0

其中 BUSY=1，显示模块内部控制器忙，不能进行操作，只有 BUSY=0 才允许进行操作。ON/OFF=1，显示关闭，ON/OFF=0，显示打开。RESET=1，复位状态，RESET=0，正常状态。在 BUSY 和 RESET 状态下，除读忙标志指令外，其他指令均不对液晶显示模块产生作用。

2．写指令

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	0	下降沿	指令							

3．写数据

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
1	0	下降沿	显示数据							

操作时每完成一个列地址，计数器自动加 1。

4．显示开/关

编码格式为

RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	1	1	1	1	D

其中 D=1，显示 RAM 中的内容；D0＝0，关闭显示。

5．显示起始行

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	1	显示起始行(0～63)						

该指令规定显示屏上起始行对应 DDRAM 的行地址，有规律地改变显示起始行，可以实现显示滚屏的效果。

6．页面地址

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	0	1	1	1	页面(0～7)			

DDRAM 共 64 行，分 8 页，每页 8 行。

7．列地址

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	显示列地址(0～63)						

列地址计数器在每一次读/写数据后自动加 1，每次操作后明确起始列的地址。设置了页面地址和列地址，就唯一确定了 DDRAM 中的一个单元，这样就可以用读/写指令读出该单元中的内容，或向该单元写进一个字节数据。

8．读数据

编码格式为

RS	R/W	E	D7	D6	D5	D4	D3	D2	D1	D0
1	1	显示数据								

该指令将 DDRAM 对应单元中的内容读出，然后列地址计数器自动加 1。需要注意的是，进行读操作之前，必须有一次空读操作，紧接着再读，才会读出所要求单元中的数据。

例 5-2 图 5-4 所示为 LPC2138 与 12864 图型液晶显示模块的一种接口电路。液晶显示模块的 E、RW、RS、$\overline{CS2}$ 和 $\overline{CS1}$ 信号分别由 LPC2138 的 P0.8、P0.9、P0.10、P0.11 和 P0.12 控制，P0.0～P0.7 用作显示模块的 8 位数据线。

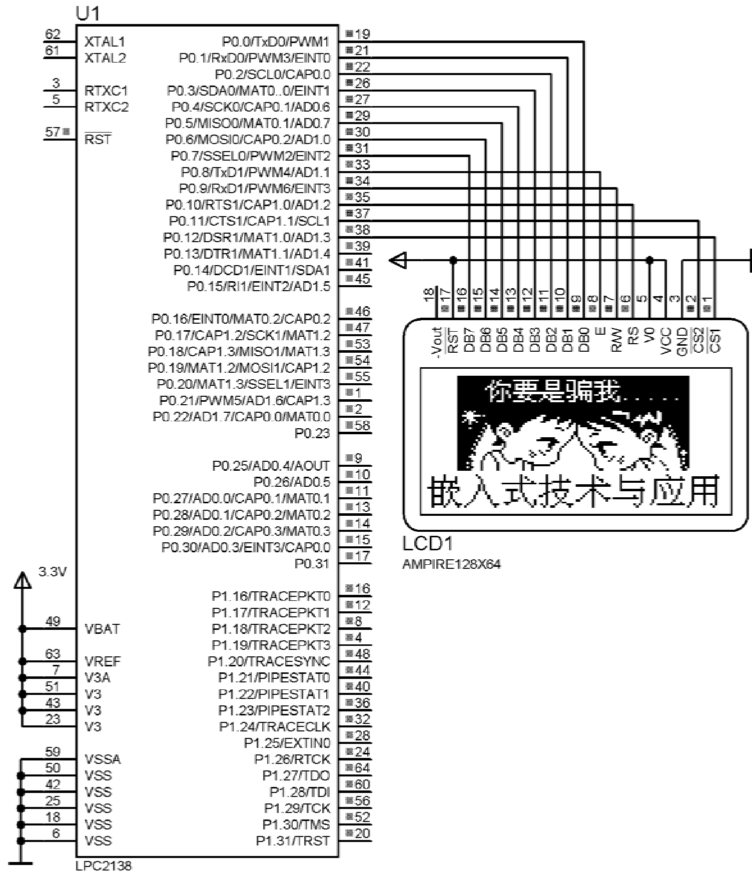


图 5-4　LPC2138 与 12864 点阵图型液晶显示模块的接口电路

本例的液晶显示驱动程序包括在 LCD12864.C 模块中，具体如下：

```
#include   "config.h"

#define E     (1<<8)
#define RW   (1<<9)
#define RS   (1<<10)
#define CS2 (1<<11)
#define CS1 (1<<12)

/**************************************************************************
*函数名称: DelayNS()
*函数功能: 长软件延时
*入口参数   dly：延时值
**************************************************************************/
void DelayNS(uint32 dly){
    uint32   i;
    for (;dly>0;dly--)
        for(i=0;i<50;i++);
}

/**************************************************************************
*函数名称: Left()
*函数功能: 左半屏
**************************************************************************/
void Left(){
      IO0CLR=CS1;    IO0SET=CS2;            //CS1=0; CS2=1
}

/**************************************************************************
*函数名称: Right()
*函数功能: 右半屏
**************************************************************************/
void Right(){
      IO0CLR=CS2;    IO0SET=CS1;            //CS1=1; CS2=0;
}

/**************************************************************************
*函数名称: SelectScreen()
*函数功能: 左、右半屏选择
*入口参数   screen=1 选择右半屏，screen=0 选择左半屏
**************************************************************************/
void SelectScreen(uint8 screen){
    switch(screen){
        case 1:
            Right();      break;      //右屏
        case 2:
```

```
            Left();break;               //左屏
        default:
            break;
    }
}

/****************************************************************************
*函数名称：ChekBusy()
*函数功能：查忙状态
****************************************************************************/
void CheckBusy(void){
      uint32    state;
      PINSEL0=0x00000000;
      IO0DIR=0x1fff;
      IO0CLR=RS;
      IO0SET=RW;
      do{
            IO0DIR=0x1fff;
            IO0CLR=0xff;
            IO0SET=E;
            IO0DIR=0x1f00;
            state=IO0PIN;
            IO0CLR=E;
      }while(0x80&state);
    IO0DIR=0x1fff;
}

/****************************************************************************
*函数名称：Wr_COM()
*函数功能：写命令
*入口参数：wdate    待写入的命令
****************************************************************************/
void    Wr_COM(uint8 wdate){
      CheckBusy();
      IO0CLR=RW;
      IO0CLR=RS;
      IO0SET=wdate;               //写入命令
      IO0SET=E;
      IO0CLR=E;
      DelayNS(1);
      IO0SET=E;
}

/****************************************************************************
*函数名称：Wr_Dat()
*函数功能：写数据
*入口参数：wdate    待写入的数据
```

```
*************************************************************************/
void Wr_Dat(uint8 wdate){
    CheckBusy();
    IO0CLR=RW;
    IO0SET=RS;
    IO0SET=wdate;                   //写数据
    IO0SET=E;
    IO0CLR=E;
    DelayNS(1);
    IO0SET=E;
}

/*************************************************************************
*函数名称：SetLine()
*函数功能：设定行(页)
*入口参数：line   行(页)号，0<=line<=7
*************************************************************************/
void SetLine(uint8 line){
    line &= 0x07;
    line |= 0xb8;
    Wr_COM(line);
}

/*************************************************************************
*函数名称：SetColumn()
*函数功能：设定列
*入口参数：column   列号，0=<column<=63
*************************************************************************/
void SetColumn(uint8 column){
    column &= 0x3f;
    column |= 0x40;
    Wr_COM(column);
}

/*************************************************************************
*函数名称：Dsp()
*函数功能：显示图片
*入口参数：lin 起始行，column 起始列，high 高，width 宽，*address 图片数组指针
*************************************************************************/
void Dsp(uint8 lin, uint8 column, uint8 high, uint8 width, uint8 *address){
    uint8 i, j;
    if(column<64){
        for(j=0;j<high;j+=8){
            SelectScreen(2);                //如果列数<64，则从第一屏上开始写
            SetLine(lin);
            SetColumn(column);
```

```
                for(i=0;i<width;i++){
                    if(i+column>127) break;
                    if(column+i<64){
                        Wr_Dat(*(address+i));
                    }
                    else{
                        SelectScreen(1);
                        SetLine(lin);
                        SetColumn(column-64+i);
                        Wr_Dat(*(address+i));
                    }
                }
                lin+=1;
                address+=width;
            }
        }
        else{
            column-=64;                              //防止越界
            for(j=0;j<high;j+=8){
                SelectScreen(2);                     //否则从第二屏上开始写
                SetLine(lin);
                SetColumn(column);
                for(i=0;i<width;i++){
                    if(i+column>64) break;
                    if(column+i<64){
                        Wr_Dat(*(address+i));
                    }
                    else{
                        SelectScreen(1);
                        SetLine(lin);
                        SetColumn(column-64+i);
                        Wr_Dat(*(address+i));
                    }
                }
                lin+=1;
                address+=width;
                }
        }
}

/****************************************************************************
*函数名称：LCD_init()
*函数功能：LCD 初始化
****************************************************************************/
void LCD_init(void){
        uint8 i,j;
        Left();
```

```
        Wr_COM(0x3f);                       //开显示
        Right();
        Wr_COM(0x3f);
        Left();                             //左半屏
        for(i=0;i<8;i++){
                Wr_COM(0xb8|i);             //设置页地址
                Wr_COM(0x40|0x00);          //设置列地址
                for(j=0;j<64;j++){
                        Wr_Dat(0x00);
                }
        }
        Right();                            //右半屏
        for(i=0;i<8;i++){
                Wr_COM(0xb8|i);             //设置页地址
                Wr_COM(0x40|0x00);          //设置列地址
                for(j=0;j<64;j++){
                        Wr_Dat(0x00);
                }
        }
}
/***************************************************************************
*函数名称：display_word()
*函数功能：显示一个汉字
入口参数： page 页地址，row  列地址， *P 汉字数组指针
***************************************************************************/
void display_word(uint8 page,uint8 row, uint8 *p){
        uint8 i;
        Wr_COM(0xb8|page);                  //设置页地址
        Wr_COM(0x40|row);                   //设置列地址
        for(i=0;i<16;i++){                  //16*16 汉字，先写上半部分
                        Wr_Dat(*p); p++;
        }
        Wr_COM(0xb8|(page+1));              //页地址加 1
        Wr_COM(0x40|row);                   //设置列地址
        for(i=0;i<16;i++){
                Wr_Dat(*p); p++;            //写下半部分
        }
}
```

进入主程序后，首先调用液晶模块初始化子程序，初始化内容包括开显示、分别对左、右半屏设置页地址和列地址，并写入初始数据 0x00，进行清屏，然后调用子程序，显示一幅图片，接下来再调用子程序，分别在左、右半屏上显示汉字。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>
```

```
#include    "pic.h"

/*****************************************************************************
*名称：main()
*功能：显示图片和汉字
*****************************************************************************/
 int main (void) {
        LCD_init();
        Dsp(0, 13, 46, 100,Pic);                //显示图片
        Left();
        display_word(6,0, dan);                 //显示汉字
        display_word(6,16, dan+32);
        display_word(6,32, dan+64);
        display_word(6,48, dan+96);
        Right();
        display_word(6,64, dan+128);
        display_word(6,80, dan+160);
        display_word(6,96, dan+192);
        display_word(6,112, dan+224);
}
```

5.1.3　T6963C 点阵图型液晶显示模块接口技术

内置 T6963C 控制器的 LCD 模块是目前较为常用且品种较多的一类点阵图形液晶显示模块，T6963C 是日本东芝公司的产品，它最大的特点是具有硬件初始值设置功能，其初始化工作在加电时就已经基本完成，软件操作的主要精力可以全部用于显示画面的设计上。T6963C 内置有 128 种 5×8 点阵的 ASCII 字符发生器 CGROM，并允许在显示存储器内开辟一个用户自定义的 8×8 点阵字模库 CGRAM。T6963C 可以管理 64KB 的显示存储器，它可以把显示存储器分成文本显示区、图形显示区以及自定义字符库区等。

图 5-5 所示为内置 T6963C 驱动器的点阵图形液晶显示模块引脚排列，引脚功能如表 5-9 所列。

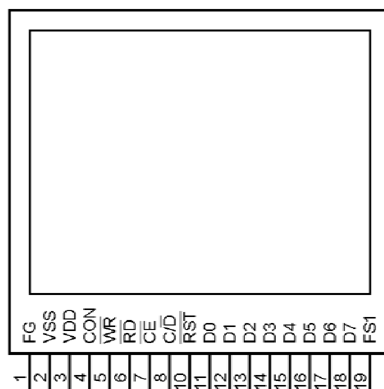


图 5-5　内置 T6963C 的点阵图型液晶显示模块引脚排列

表 5-9　内置 T6963C 的点阵图型液晶显示模块的引脚功能

引　　脚	符　　号	功　　能	引　　脚	符　　号	功　　能
1	FG	显示屏框架外壳地接地	7	$\overline{CE}$	片选端，低电平有效
2	VSS	电源地	8	$C/\overline{D}$	通道选择端，1 为命令通道，0 为数据通道
3	VDD	+5V 电源	10	$\overline{RST}$	复位信号，1 为正常工作，0 为初始化 T6963C
4	CON	对比度调节	11～18	D0～D7	数据信号
5	$\overline{WR}$	数据写入，低电平有效	19	FS1	字体选择，1 为选择 6×8 点阵字体，0 为选择 8×8 点阵字体
6	$\overline{RD}$	数据读出，低电平有效			

T6963C 提供两种命令形式：带参数命令和无参数命令。

带参数命令中的参数需要在命令编码之前输入，格式如下：

参数 1　参数 2　命令编码

无参数命令只要给出命令编码即可。表 5-10 列出了 T6963C 的全部命令编码。

表 5-10　T6963C 的命令编码

命　　令	参数 1	参数 2	编　　码	功　　能
寄存器设置	水平位置（低 7 位有效）	垂直位置（低 5 位有效）	0010 0001	设置光标位置
	偏置地址（低 5 位有效）	00H	0010 0010	设置 CGRAM 偏置地址
	地址低 8 位	地址高 8 位	0010 0100	设置显示地址
显示区域设置	地址低 8 位	地址高 8 位	0100 0000	设置文本起始地
	列	00H	0100 0001	设置文本区宽度
	地址低 8 位	地址高 8 位	0100 0010	设置图形起始地址
	列	00H	0100 0011	设置图形区宽度
模式设定	—	—	1000 x000	文本与图形以“或”关系合成显示
	—	—	1000 x001	文本与图形以“异或”关系合成显示
	—	—	1000 x010	文本与图形以“与”关系合成显示
	—	—	1000 x011	文本显示特征以双字节表示
	—	—	1000 0xxx	内部 CGROM 模式
	—	—	1000 1xxx	外部 CGRAM 模式
显示模式	—	—	1001 0000	显示关闭
	—	—	1001 xx10	打开光标，黑色关闭
	—	—	1001 xx11	打开光标，黑色显示
	—	—	1001 01xx	打开文本方式，关闭图形方式
	—	—	1001 10xx	关闭文本方式，打开图形方式
	—	—	1001 11xx	图形文本混合方式

（续）

命　令	参数 1	参数 2	编　码	功　能
光标形式	—	—	1010 0000	1 条线
	—	—	1010 0001	2 条线
	—	—	1010 0010	3 条线
	—	—	1010 0011	4 条线
	—	—	1010 0100	5 条线
	—	—	1010 0101	6 条线
	—	—	1010 0110	7 条线
	—	—	1010 0111	8 条线
数据自动读写	—	—	1011 0000	数据自动写入设定
	—	—	1011 0001	数据自动读出设定
	—	—	1011 0000	自动复位
数据读写	—	—	1100 0000	数据写入，地址自动增量
	—	—	1100 0001	数据读出，地址自动增量
	—	—	1100 0010	数据写入，地址自动减量
	—	—	1100 0011	数据读出，地址自动减量
	—	—	1100 0100	数据写入，地址保持不变
	—	—	1100 0101	数据读出，地址保持不变
屏幕读取	—	—	1110 0000	读取屏幕显示数据
屏幕复制	—	—	1110 1000	复制屏幕显示数据
位操作	—	—	1111 0xxx	位清零
	—	—	1111 1xxx	位置 1
	—	—	1111 x000	位 0
	—	—	1111 x001	位 1
	—	—	1111 x010	位 2
	—	—	1111 x011	位 3
	—	—	1111 x100	位 4
	—	—	1111 x101	位 5
	—	—	1111 x110	位 6
	—	—	1111 x111	位 7

注：表中参数栏的“—”表示无参数。

T6963C 提供一个状态字，格式如下：

S7	S6	S5	S4	S3	S2	S1	S0

其中各位的含义如下：

S0：命令读写状态，S0=1 为准备好，S0=0 为忙。

S1：数据读写状态，S1=1 为准备好，S1=0 为忙。

S2：数据自动读状态，S2=1 为准备好，S2=0 为忙。

S3：数据自动写状态，S3=1 为准备好，S3=0 为忙。

S4：未用。

S5：控制器运行检测可能性，S5=1 为可能，S5=0 为不能。

S6：屏幕读取 / 屏幕复制出错状态，S6=1 为出错，S6=0 为正确。

S7：闪烁状态检测，S7=1 为显示，S6=0 为关闭。

这些标识位各有各的应用场合，并非同时有效。CPU 在写命令一次读写数据时，S0 和 S1 要同时有效，即“准备好”状态。当 CPU 采用自动读写功能时，S2 或 S3 将取代 S0 和 S1 作为忙标志。S6 是考察 T6963C 屏幕读取和屏幕复制命令执行情况的标志位。S5 和 S7 表示控制器内部的运行状态，T6963C 应用时不会使用它们。对 T6963C 进行每一次软件操作之前，都要判读忙标志，只有在不忙（即“准备好”）状态下，CPU 对 T6963C 的操作才有效。

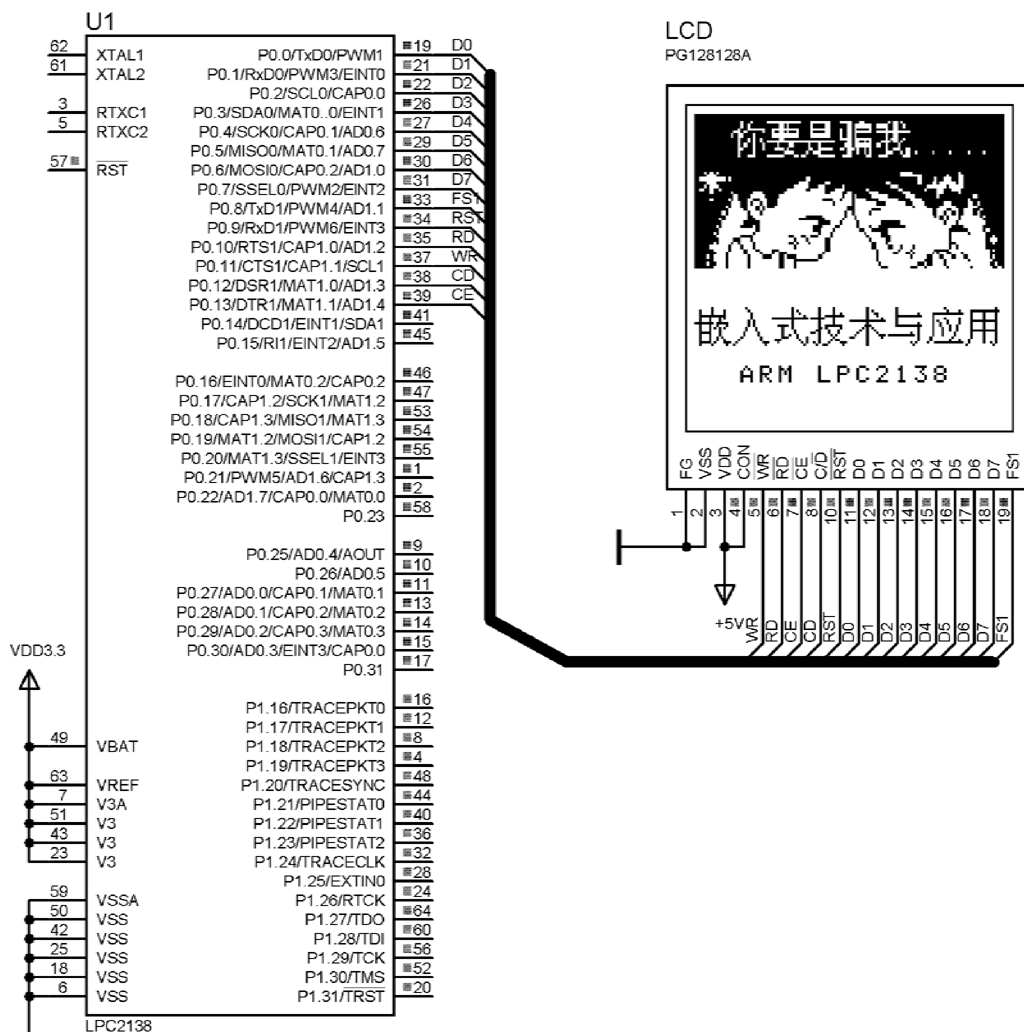


图 5-6　LPC2138 与 T6963C 点阵图型液晶显示模块的接口电路

例 5-3　图 5-6 所示为 LPC2138 与 T6983C 图型液晶显示模块的一种接口电路。液晶显示模块的 FS1、$\overline{RST}$、$\overline{RD}$、$\overline{WR}$、C/$\overline{D}$ 和 $\overline{CE}$ 信号分别由 LPC2138 的 P0.8、P0.9、P0.10、P0.11、P0.12 和 P0.13 控制，P0.0～P0.7 用作显示模块的 8 位数据线。

本例的液晶显示驱动程序包括在 T6963.C 模块中，具体如下：

```
#include "Config.h"
#include "t6963.h"
#include "dat.h"

/**************************************************************************
* 函数名称：wr_od()
* 函数功能：写一个数据和一个命令
* 入口参数：dat      待写入的数据，comm   待写入的命令
**************************************************************************/
void wr_od (uint8 dat,uint8 comm){          //写一个数据和一个命令
   wr_data(dat);
   wr_comm(comm);
}

/**************************************************************************
* 函数名称：wr_td()
* 函数功能：写两个数据和一个命令
* 入口参数：dat1 待写入的第一个数据，dath 待写入的第二个数据，comm 待写入的命令
**************************************************************************/
void wr_td (uint8 datl,uint8 dath,uint8 comm){
   wr_data(datl);
   wr_data(dath);
   wr_comm(comm);
}

/**************************************************************************
* 函数名称：wr_xd()
* 函数功能：写一个十六进制数据和一个命令
* 入口参数：dat 待写入的十六进制数据，comm 待写入的命令
**************************************************************************/
void wr_xd (uint16 dat,uint8 comm){
   wr_data(dat);
   wr_data(dat>>8);
   wr_comm(comm);
}

/**************************************************************************
* 函数名称：wr_auto()
* 函数功能：自动写数据
* 入口参数：dat 待写入的数据
**************************************************************************/
void wr_auto (uint8 dat){
   chk_busy (1);
   IO0CLR=cd;
   IO0SET=rd;
```

```
    IO0PIN=(IO0PIN&0XFFFFFF00|dat);
    IO0CLR=wr;
    IO0SET=wr;
}

/******************************************************************************
* 函数名称：wr_comm()
* 函数功能：写命令
* 入口参数：dat 待写入的命令
******************************************************************************/
void wr_comm (uint8 comm){
    chk_busy (0);
    IO0SET=cd;
    IO0SET=rd;
    IO0PIN=(IO0PIN&0XFFFFFF00|comm);
    IO0CLR=wr;
    IO0SET=wr;
}

/******************************************************************************
* 函数名称：wr_data()
* 函数功能：写数据
* 入口参数：dat 待写入的数据
******************************************************************************/
void wr_data (uint8 dat){
    chk_busy (0);
    IO0CLR=cd;
    IO0SET=rd;
    IO0PIN=(IO0PIN&0XFFFFFF00|dat);
    IO0CLR=wr;
    IO0SET=wr;
}

/******************************************************************************
* 函数名称：chk_busy()
* 函数功能：测状态
* 入口参数：autowr 自动写数据时为 1，否则为 0
******************************************************************************/
void chk_busy (uint8 autowr){
    IO0SET=0xff;
    IO0SET=cd;
    IO0SET=wr;
    IO0CLR=rd;
    IO0DIR &= 0xFFFFFF00;
    if(autowr)
        {while((IO0PIN&bf3)==0)
```

```
         ;}
   else
      {while((IO0PIN&bf0)==0||(IO0PIN&bf1)==0)
         ;}
   IO0SET=rd;
   IO0DIR |= 0x000000FF;
}

/*******************************************************************************
* 函数名称：clrram()
* 函数功能：清除显示 RAM
*******************************************************************************/
void clrram (void){
   uint8 i,j;
   wr_xd(addr_w,0x24);
   wr_comm(0xb0);
   for(j=0;j<144;j++){
      for(i=0;i<width;i++)
         wr_auto(0x00);
   }
   wr_comm(0xb2);
}

/*******************************************************************************
* 函数名称：disp_img()
* 函数功能：显示图形
* 入口参数：addr 显示 RAM 地址，xl 图宽，yl 图高，*img 图形数据指针
*******************************************************************************/
void disp_img (uint8 addr,uint8 xl,uint8 yl,uint8 const *img){
   uint8 i,j;
   for(j=0;j<yl;j++){
      for(i=0;i<xl;i++){
         wr_xd(addr+0X100+j*width+i,0x24);
         wr_od(img[j*xl+i],0xc0);
      }
   }
}

/*******************************************************************************
* 函数名称：strlen1()
* 函数功能：计算字符串长度
* 入口参数：*s 字符串指针
*******************************************************************************/
int strlen1(char *s){
        int i=0;
        while(s[i])i++;
```

```
        return i;
}

/***************************************************************************
*  函数名称：disp_hz()
*  函数功能：显示汉字
*  入口参数：x 列位置，y  行位置，*str  汉字数据指针
****************************************************************************/
void disp_hz(uint32 x,uint32 y, char *str){
        uint8 i,j,n,c1,c2,slen,m=0;
        uint32 tempx,tempy;
        tempx=x;
        tempy=y;
         slen = strlen1(str);
         while(m<slen-1){
                c1=str[m+0];c2=str[m+1];
                for(n=0;n<sizeof(GB_16)/sizeof(GB_16[0]);n++){
                      if(c1 == GB_16[n].Index[0]&&c2 == GB_16[n].Index[1])
                      break;
                }
                y=tempy;
                x=tempx;
                for(i=0;i<16;i++){
                      for (j=0;j<2;j++){
                            wr_xd((0x0100+256*y+16*i+x+j),0x24);
                            wr_od(GB_16[n].Msk[i*2+j],0xc0);
                      }
                }
                m=m+2;//一个汉字两个字节，LCDSIZE/8
                tempx += 2;
                if(tempx>15){
                       tempx = 0;
                       tempy += 1;
                }
         }
}

/***************************************************************************
*  函数名称：disp_zf()
*  函数功能：显示字符
*  入口参数：x 列位置，y  行位置，*str  字符数据指针
****************************************************************************/
void disp_zf(uint32 x,uint32 y,uint8 *str){
         char c;
         wr_xd((addr_w+16*y+x),0x24);
         wr_comm(0xb0);
         while(*str!='\0'){
```

```
            c = (*str);
            wr_auto(c-32);
            str++;
        }
        wr_comm(0xb2);
    }
```

进入主程序后首先进行引脚配置，接着调用液晶模块初始化子程序，初始化内容包括设置文本显示区首地址、图形显示区首地址、文本显示区宽度、图形显示区宽度、CGRAM 偏置地址，然后启用文本和图形。初始化完成之后进行清屏，最后分别调用图形显示子程序、汉字显示子程序以及字符显示子程序，完成整个屏幕的显示。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
#include "t6963.h"

/************************************************************************
*  函数名称：DelayNS()
*  函数功能：长软件延时
*  入口参数：dly       延时参数，值越大，延时越久
************************************************************************/
void   DelayNS(uint32   dly){   uint32   i;
    for(; dly>0; dly--)
        for(i=0; i<50000; i++);
}

/************************************************************************
*  函数名称：init_lcd()
*  函数功能：LCD 初始化
************************************************************************/
void init_lcd (void){
  IO0CLR=rst;
  IO0SET=rst;
  DelayNS(5);
  IO0CLR=ce;
  IO0SET=wr;
  IO0SET=rd;
  wr_xd(addr_w,0x40);                  // 文本显示区首地址
  wr_xd(addr_t,0x42);                  // 图形显示区首地址
  wr_td(width,0x00,0x41);              // 文本显示区宽度
  wr_td(width,0x00,0x43);              // 图形显示区宽度
  wr_comm(0x81);                       // 逻辑"异或"
  wr_td(0x56,0x00,0x22);               // CGRAM 偏置地址设置
  wr_comm(0x9c);                       // 启用文本显示，启用图形显示
}
```

```
/****************************************************************************
* 函数名称：main()
* 函数功能：LCD 显示图片和汉字
****************************************************************************/
int main (){
    PINSEL0 = 0x0000000;                    // 引脚配置
    PINSEL1 = 0x00000000;
    IO0DIR =    0xffffffff;
    init_lcd ();                            // LCD 初始化
    clrram();                               // 清屏
    disp_img(0,16,64,nBitmapDot);           // 显示图画
    disp_hz(0,5, "嵌入式技术与应用");          // 显示汉字
    disp_zf(0,13, "    ARM LPC2138    ");   // 显示字符
    while(1);
}
```

5.2 用 DS18B20 与 LPC2138 实现的数字温度计

DS18B20 是一种新型数字温度传感器，它采用独特的单线接口方式，仅需一个端口引脚来发送或接收信息，在 CPU 和 DS1B820 之间仅需一条数据线和一条地线进行接口。DS1B820 采用 TO-92 或 8 脚 SOIC 封装，引脚排列如图 5-7 所示。各引脚功能如下：

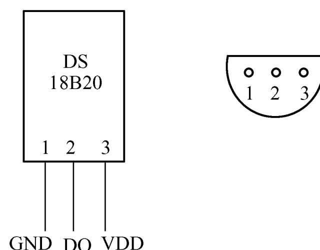


图 5-7 DS18B20 引脚排列

- GND：地。
- DQ：单线应用的数据输入/输出引脚。
- VDD：可选的外部供电电源引脚。

DS18B20 内部有三个主要数字部件：64 位激光 ROM、温度传感器、非易失性温度报警触发器 TH 和 TL。DS1B820 可以采用寄生电源方式工作，从单总线上汲取能量，在信号线处于高电平期间，把能量储存在内部电容里，在信号线处于低电平期间，消耗电容上的电能工作，直到高电平到来再给寄生电源（电容）充电。DS18B20 也可用外部 3～5.5V 电源供电，这两种供电方式的电路如图 5-8 所示。

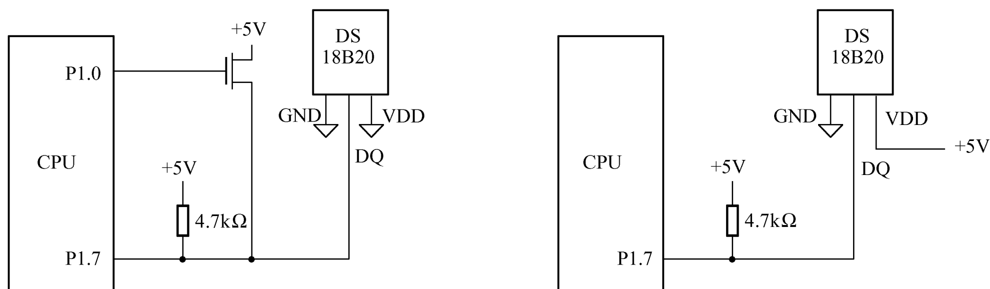


图 5-8 DS18B20 的供电方式

采用寄生电源方式时，VDD 引脚必须接地，另外为了得到足够的工作电流，应给 CPU 的 I/O 口线提供一个强上拉，一般可以使用一个场效应晶体管，将 I/O 口线直接拉到电源上。采用外部供电方式时，可以不用强上拉，但外部电源要处于工作状态，GND 引脚不得悬空。温度高于 100℃时，不推荐使用寄生电源，应采用外部电源供电。

DS18B20 依靠一个单线端口通信，必须先建立 ROM 操作协议，才能进行存储器和控制操作。因此，CPU 必须先提供下面 5 个 ROM 操作命令之一：

① 读出 ROM，代码为 33H，用于读出 DS18B20 的序列号，即 64 位激光 ROM 代码。

② 匹配 ROM，代码为 55H，用于辨识（或选中）某一特定的 DS18B20 进行操作。

③ 搜索 ROM，代码为 F0H，用于确定总线上的节点数以及所有节点的序列号。

④ 跳过 ROM，代码为 CCH，命令发出后系统将对所有 DS18B20 进行操作，通常用于启动所有 DS18B20 转换之前，或系统中仅有一个 DS18B20 时。

⑤ 报警搜索，代码为 ECH，主要用于鉴别和定位系统中超出程序设定的报警温度界限的节点。

这些命令对每个器件的激光 ROM 部分进行操作，在单线总线上挂有多个器件时，可以区分出单个器件，同时指明有多少器件或是什么型号的器件。

DS18B20 内部存储器映像如图 5-9 所示。存储器由一个高速暂存器和一个存储高低温报警触发值 TH 和 TL 的非易失性电可擦除 E^2RAM 组成。头 2 个字节为实测温度值，低字节在前，高字节在后，第 3 和第 4 字节是用户设定温度报警值 TH 和 TL 的拷贝，是易失的，每次上电时被刷新。第 5 个字节为配置寄存器，其内容用于确定温度值的数字转换分辨率，DS18B20 工作时，按此寄存器中的分辨率，将温度转换为相应精度的数值。

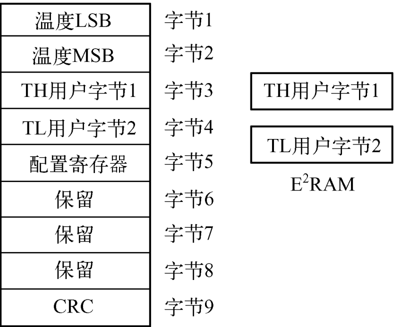


图 5-9　DS18B20 内部存储器映像

配置寄存器各位的分布如下：

D7	D6	D5	D4	D3	D2	D1	D0
TM	R1	R0	1	1	1	1	1

其中，TM 为测试模式位，用于设定 DS18B20 是处于工作模式还是处于测试模式，出厂时 TM 位被设置为 0，用户一般不要改动。R1 和 R0 用于设定温度转换的精度分辨率，如表 5-11 所列。其余低 5 位全为 1。DS18B20 温度转换时间较长，而且设定的分辨率越高，所需转换时间越长，因此实际应用中，要根据具体情况权衡考虑。

表 5-11　DS18B20 的分辨率设定

R1	R0	分辨率/位	温度最大转换时间/ms
0	0	9	93.75
0	1	10	187.5
1	0	11	375
1	1	12	750

高速暂存器的第 6、7、8 字节保留未用，读出值为全 1。第 9 字节为前面 8 个字节的 CRC 校验码，用于保证数据通信的正确性。

DS18B20 提供了如下存储器操作命令：

① 温度转换，代码为 44H，用于启动 DS18B20 进行温度测量。温度转换命令被执行后，DS18B20 保持等待状态。如果主机在这条命令之后跟着发出读时间隙，而 DS18B20 又忙于进行温度转换的话，DS18B20 将在总线上输出“0”，若温度转换完成，则输出“1”。如果使用寄生电源，主机必须在发出这条命令后立即启动强上拉，并保持 750ms，在这段时间内单总线上不允许进行任何其他操作。

② 读暂存器，代码为 BEH，用于读取暂存器中的内容。从字节 0 开始最多可以读取 9 个字节，如果不想读完所有字节，主机可以在任何时间发出复位命令来中止读取。

③ 写暂存器，代码为 4EH，用于将数据写入到 DS18B20 暂存器的地址 2 和地址 3（TH 和 TL 字节）。可以在任何时刻发出复位命令来中止写入。

④ 复制暂存器，代码为 48H，用于将暂存器的内容复制到 DS18B20 的非易失性 E^2RAM 中，即把温度报警触发字节存入非易失性存储器里。如果主机在这条命令之后跟着发出读时间隙，而 DS18B20 又正在忙于把暂存器的内容复制到 E^2RAM 存储器，DS18B20 就会输出一个“0”，如果复制结束的话，DS18B20 则输出“1”。如果使用寄生电源，主机必须在这条命令发出后，立即启动强上拉并最少保持 10ms，在这段时间内，单总线上不允许进行任何其他操作。

⑤ 重读 E^2RAM，代码为 B8H，用于将存储在非易失性 E^2RAM 中的内容重新读入到暂存器（温度触发器）中。这种复制操作在 DS18B20 上电时自动执行，这样，器件一上电，暂存器里马上就存在有效的数据了。若在这条命令发出之后发出读时间隙，器件会输出温度转换忙的标志：“0”代表忙，“1”代表完成。

⑥ 读电源，代码为 B4H，用于将 DS18B20 的供电方式信号发送到主机。若在这条命令发出之后发出读时间隙，DS18B20 将返回它的供电模式：“0”代表寄生电源，“1”代表外部电源。

一条温度转换命令启动 DS18B20 完成一次温度测量，测量结果以二进制补码形式存放在的高速暂存器中，占用暂存器的字节 1（LSB）和字节 2（MSB）。用一条读暂存器内容的存储器操作命令，可以把暂存器中的数据读出。温度报警触发器 TH 和 TL 各由一个 E^2PROM 字节构成，可以用一条写存储器操作命令对 TH 和 TL 进行写入，对这些寄存器的读出需要通过暂存器。所有数据都以低位（LSB）在前的方式进行读写，数据格式以 0.0625℃/LSB 形式表示如下：

LSB 字节

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

MSB 字节

S	S	S	S	S	2^6	2^5	2^4

当符号位 S=0 时，表示测得的温度值为正，可以直接对测得的二进制数进行计算，并转换为十进制数。当符号位 S=1 时，表示测得的温度值为负，此时测得的二进制数为补码数，要先变成原码数再进行计算。表 5-12 所示为部分温度值对应二进制数据。

表 5-12　DS18B20 温度与测量值对应表

温度/℃	二进制数表示	十六进制数表示
+125	00000111 11010000	07D0H
+85	00000101 01010000	0550H
+25.065	00000001 10010001	0191H
+10.125	00000000 10100010	00A2H
+0.5	00000000 00001000	0008H
0	00000000 00000000	0000H
−0.5	11111111 11111000	FFF8H
−10.125	11111111 01011110	FF5EH
−25.0625	11111110 01101111	FE6FH
−55	11111100 10010000	FC90H

DS18B20 完成温度转换后，就把测得的温度值 t 与暂存器中的 TH、TL 字节内容进行比较，若 t >TH 或 t < TL，则将 DS18B20 内部报警标志位置 1，并对主机发出的报警搜索命令做出响应，因此可用多只 DS18B20 进行多点温度循环监测。

例 5-4　图 5-10 所示为 LPC2138 与 DS18B20 组成的数字温度计电路。DS18B20 采用外部 3.3V 电源供电，LPC2138 的 P1.25 引脚外接一个 4.7kΩ上拉电阻，连到 DS18B20 的 DQ 端作为数据线。温度转换结果通过点阵字符型 LCD 显示。

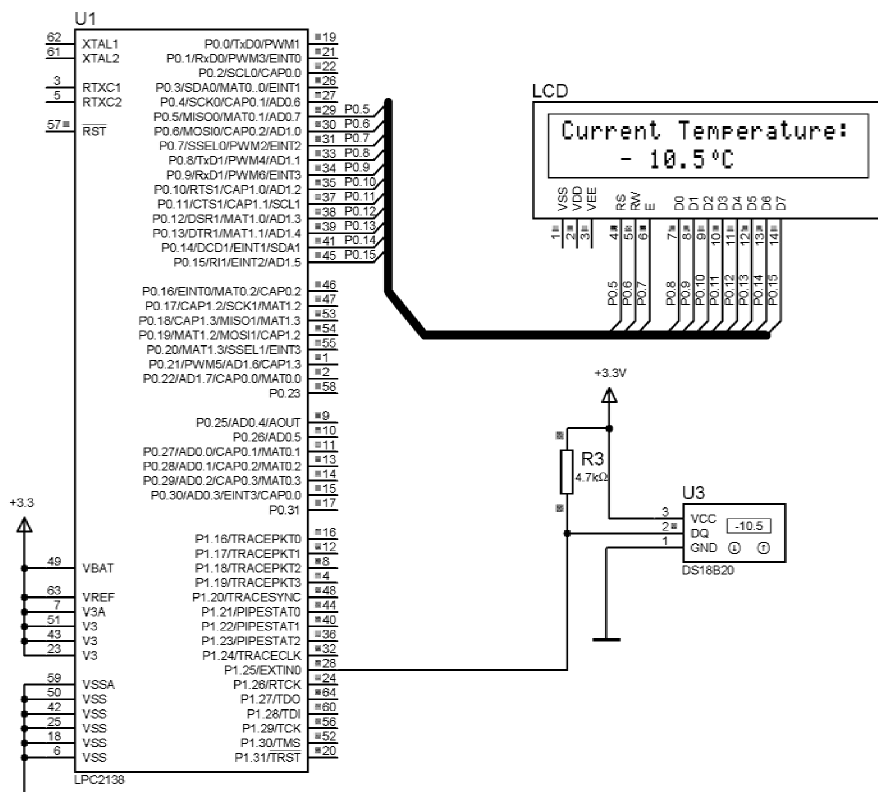


图 5-10　LPC2138 与 DS18B20 组成的数字温度计电路

本例中 DS18B20 驱动程序包括在 DS18B20.C 模块中，具体如下：

```
#include "config.h"
#include "ds18b20.h"

#define DQ          (1<<25)             //定义 DS18B20 通信端口

uint8 zfflag=0;                         //温度正负标志
uint8 err;                              //DS18B20 检测标志
uint8 Terr[]={"T-Err"};
uint8 temp_value,temp_value2;           //温度值

/************************************************************************
* 函数名称：Delay_DS18B20()
* 函数功能：软件延时
* 入口参数：time 延时参数，值越大，延时越久
************************************************************************/
void Delay_DS18B20(uint32 time){
    uint32 i,j;
    for(i=0;i<8;i++){
        for(j=1;j<time;j++);
    }
}

/************************************************************************
* 函数名称：Init_DS18B20()
* 函数功能：DS18B20 初始化
* 出口参数：x=0 则初始化成功；x=1 则初始化失败
************************************************************************/
uint8 Init_DS18B20(void){
    uint8 x=1;
    IO1DIR|=DQ;                 //设置引脚为输出
    IO1SET|=DQ;                 //DQ 复位
    Delay_DS18B20(2);           //稍作延时
    IO1CLR|=DQ;                 //将 DQ 拉低
    Delay_DS18B20(560);         //精确延时 大于 480µs
    IO1SET|=DQ;                 //拉高总线
    IO1DIR&=(～DQ);             //转为输入
    Delay_DS18B20(55);          //延迟 30～130µs
    x=(IO1PIN&DQ)>>25;          //150µs 延时后，如果 x=0，则初始化成功；x=1 则初始化失败
    Delay_DS18B20(480);         //精确延时  110～480µs
    return (x);
}

/************************************************************************
* 函数名称：ReadOneChar()
* 函数功能：DS18B20 读一个字节
* 出口参数：dat 读取的数据
```

```
*****************************************************************************/
uint8 ReadOneChar(void){
    uint8 i=0;
    uint8 dat = 0;
    IO1DIR|=DQ;               //设置引脚为输出
    for (i=8;i>0;--i){
        dat>>=1;
        IO1DIR|=DQ;           //设置引脚为输出
        IO1SET|= DQ;          //给脉冲信号
        Delay_DS18B20(2);
        IO1CLR|=DQ;           //给脉冲信号
        Delay_DS18B20(10);    //延迟
        IO1SET|= DQ;          //给脉冲信号
        Delay_DS18B20(10);    //此时间不能太大，大了就会不稳定
        IO1DIR&=(~DQ);        //设置引脚为输入
        if(IO1PIN & DQ)dat|=0x80;
        Delay_DS18B20(60);    //至少等待 50μs
    }
    IO1DIR |= DQ;             //设置引脚为输出
    IO1SET |= DQ;             //给脉冲信号
     return(dat);
}

/*****************************************************************************
* 函数名称：WriteOneChar()
* 函数功能：DS18B20 写一个字节
* 入口参数：dat  写入的数据
*****************************************************************************/
void WriteOneChar(uint8 dat){
    uint8 i=0;
    IO1DIR|=DQ;                        //设置引脚为输出
    for (i=8; i>0; --i){
        if(dat&0x01){
            IO1CLR |= DQ;              // 给脉冲信号
            Delay_DS18B20(10);
            IO1SET|=DQ;                // 给脉冲信号
            Delay_DS18B20(60);
        }
        else{
            IO1CLR |= DQ;              // 给脉冲信号
            Delay_DS18B20(60);
            IO1SET|=DQ;                // 给脉冲信号
            Delay_DS18B20(10);
        }
       dat>>=1;
  }
}
```

```
/*****************************************************************************
* 函数名称：ReadTemp()
* 函数功能：读取 DS18B20 当前温度
* 出口参数：tvalue 读取的温度值
*****************************************************************************/
uint16 ReadTemp(void){
	uint8 a=0,b=0;
	uint16 tvalue=0;			// 温度值
	err=Init_DS18B20();
	if(!err){
		WriteOneChar(0xCC);		// 跳过读序号列号的操作
		WriteOneChar(0x44);		// 启动温度转换
		Delay_DS18B20(1000);	// 等待 500µs 延迟，以便温度转换完成
		Init_DS18B20();
		WriteOneChar(0xCC);		// 跳过读序号列号的操作
		WriteOneChar(0xBE);		// 读取温度寄存器等（共可读 9 个寄存器） 前两个就是温度
		a=ReadOneChar();		// 读取温度值低位
		b=ReadOneChar();		// 读取温度值高位
		tvalue = (b << 8);
		tvalue = tvalue | a;
		if( tvalue < 0x0FFF){		// 温度为正
		    TMark = 0;
		}
		else{				// 温度为负
			tvalue = ～tvalue + 1;
			TMark = 1;
		}
	}
	return(tvalue*(0.625));		// 温度值扩大 10 倍，精确到 1 位小数
}
```

进入主程序后首先调用初始化子程序进行引脚配置，接着进行液晶模块初始化，初始化完成之后在 LCD 屏幕上显示提示信息，接着对 DS18B20 进行预读，这是由于 DS18B20 进行温度转换需要一定时间的原因，预读期间，LCD 屏幕上没有温度值显示。最后启动 DS18B20 温度转换，并将转换结果显示在 LCD 屏幕上。

主程序文件 main.c 如下：

```
#include "config.h"
#include "DS18B20.h"

uint8   Tp[12];				// 温度显示数据
uint16 tt;
uint8 TMark = 0;			// 温度正负标志

/*****************************************************************************
* 函数名称：Init()
* 函数功能：I/O 端口初始化，设置引脚连接模块
*****************************************************************************/
void Init(void){
```

```
    PINSEL0 |= 0x00000000;                    // 设置引脚连接模块
    PINSEL1 |= 0x00000088;
    PINSEL2 &= 0x00000004;
}

/****************************************************************************
* 函数名称：ConvertTemperature()
* 函数功能：将读取的 DS18B20 温度值转换为十进制数
* 入口参数：tvalue  读取的 DS18B20 当前温度值
****************************************************************************/
void ConvertTemperature(uint16 tvalue){
    Tp[1] = tvalue/1000+0x30;                 // 百位数
    Tp[2] = tvalue%1000/100+0x30;             // 十位数
    Tp[3] = tvalue%100/10+0x30;               // 个位数
    Tp[4] = 0x2E;                             // 显示小数点
    Tp[5] = tvalue%10+0x30;                   // 显示温度小数位
    Tp[6] = 0x01;                             // 显示自定义字符，用于温度符号显示
    Tp[7]= 0x43;                              // 显示字符"C"
    Tp[8]= 0x20;                              // 显示字符"C"
    Tp[9]= 0x20;                              // 显示字符"C"
    Tp[10]= 0x20;                             // 显示字符"C"
    Tp[11]= 0x20;                             // 显示字符"C"
    if (TMark == 0){
        Tp[0] = 0x20;                         // 正温度不显示符号
    }
    else{
        Tp[0] = 0x2D;                         // 负温度显示负号:-
    }
    if (Tp[1] == 0x30){
        Tp[1] = 0x20;                         // 如果百位为 0，不显示
        if (Tp[2] == 0x30){
            Tp[2] = 0x20;                     // 如果十位为 0，不显示
        }
    }
}

/****************************************************************************
* 函数名称：DelayNS()
* 函数功能：软件延时
* 入口参数：dly  延时参数，值越大，延时越久
****************************************************************************/
void DelayNS(uint32 dly){
    uint32 i;
    for ( ; dly>0; dly--)
        for (i=0; i<50000; i++);
}

/****************************************************************************
```

```
* 函数名称：main()
* 函数功能：主函数
****************************************************************************/
int main(void){
        uint8 i;
        Init();                                          // I/O 端口初始化
        lcd_init();                                      // LCD 初始化
        DisText(0x80,"Current Temperature:");
        for(i=0;i<110;i++)ReadTemp();                    // 预读 DS18B20
        while (1){
                ConvertTemperature(ReadTemp());          // 温度转换
                DisText(0xc4,Tp);                        // 显示当前温度
        }
}
```

5.3 用 DS1302 与 LPC2138 实现的万年历

DS1302 是一种串行接口实时时钟芯片，芯片内部具有可编程日历时钟和 31 个字节的静态 RAM，日历时钟可自动进行闰年补偿，计时准确，接口简单，使用方便，工作电压范围宽（2.5～5.5V），功耗低，芯片自身还具有对备份电池进行涓流充电功能，可有效延长备份电池的使用寿命。DS1302 采用 8 脚 DIP 封装，其引脚排列如图 5-11 所示，各引脚功能如下：

V_{cc1},V_{cc2}：电源输入。

GND：地。

X1，X2：外接 32.768kHz 石英晶振输入。

$\overline{RST}$：复位端。

I/O：数据输入 / 输出端。

SCLK：串行时钟输入。

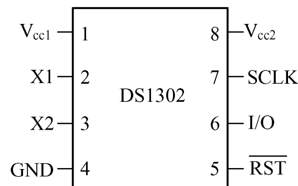


图 5-11　DS1302 的引脚排列

DS1302 芯片的 X1 和 X2 端外接 32.768kHz 的石英晶振，V_{cc1} 和 V_{cc2} 是电源引脚，单电源供电时接 V_{cc1} 脚，双电源供电时主电源接 V_{cc2}，备份电池接 V_{cc1}，如果采用可充电镉镍电池，可启用内部涓流充电器在主电压正常时向电池充电，以延长电池使用时间。备份电池也可用 1μF 以上的超容量电容代替，需要注意，备份电池电压应略低于主电源工作电压。

数据传送是以 CPU 为主控芯片进行的，每次传送时由 CPU 向 DS1302 写入一个命令字节开始。命令字节的格式如下：

D7	D6	D5	D4	D3	D2	D1	D0
1	RAM/CK	A4	A3	A2	A1	A0	RD/W

命令字节的最高位必须为 1，RAM/CK 位为 DS1302 片内 RAM/时钟选择位，RAM/CK=1 选择 RAM 操作，RAM/CK=0 选择时钟操作。RD/W 位为读写控制位；RD/W=1 为读操作，表示 DS1302 接收完命令字节后，按指定的选择对象及寄存器（或 RAM）地址，读取数据并通过 IO 线传送给 CPU，RD/W=0 为写操作，表示 DS1302 接收完命令字节

后，紧跟着再接收来自于 CPU 的数据字节，并写入到 DS1302 相应的寄存器或 RAM 单元中。A4～A0 为片内日历时钟寄存器或 RAM 的地址选择位。

DS1302 与 CPU 之间通过 I/O 线传送同步串行数据，SCLK 为串行通信时的位同步时钟，一个 SCLK 脉冲传送一位数据。每次数据传送时都以字节为单位，低位在前，高位在后，传送一个字节需要 8 个 SCLK 脉冲。数据传送可以单字节方式或多字节突发方式进行。

数据单字节方式传送时序如图 5-12 所示，在 RST=1 期间，CPU 先向 DS1302 发送一个命令字节，紧接着发送一个字节的数据，DS1302 在接收到命令字节后自动将数据写入指定的片内地址或从该地址读取数据。

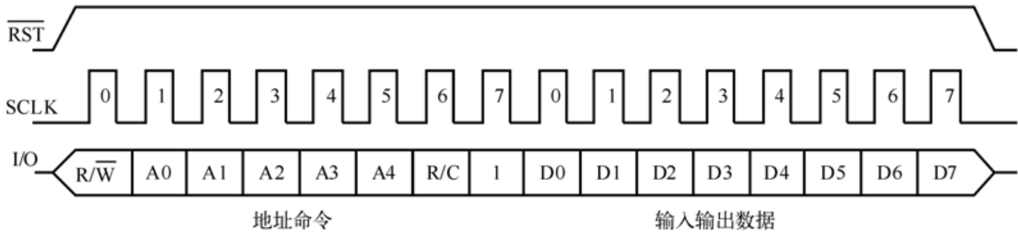


图 5-12 数据单字节方式传送时序

数据多字节突发方式传送时序如图 5-13 所示，RST=1 期间，若 CPU 向 DS1302 发送的命令字节中 A0～A4 全为 1，则 DS1302 在接收到这个字节命令后，可以一次进行 8 个字节日历时钟数据或是 31 个片内 RAM 单元数据的读写操作。

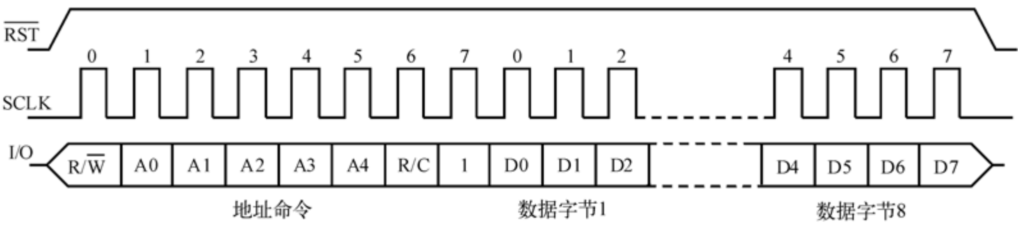


图 5-13 数据多字节突发方式传送时序

从以上时序可知，单字节方式传送一次数据需要 16 个 SCLK 脉冲，多字节方式传送一次数据，在对日历时钟进行读写时需要 72 个 SCLK 脉冲，而在对片内 RAM 单元读写时则最多需要 256 个 SCLK 脉冲。单字节操作方式可保证数据传送时的安全性和可靠性，多字节操作方式则可提高数据传送速度，两种方式可视需要灵活选用。DS1302 的外接晶振推荐采用 32.768kHz，电容推荐值为 6pF，由于晶振频率较低，也可以不接电容，对计时精度影响不大。

DS1302 共有 12 个寄存器，其中 7 个寄存器与日历时钟有关，存放的数据为 BCD 码格式，日历、时钟寄存器地址及其内容如表 5-13 所列。秒寄存器的第 7 位为时钟暂停控制位，该位为 1 时，暂停时钟振荡器，DS1302 进入低功耗状态，该位为 0 时，启动时钟。时寄存器的第 7 位为 12 或 24 小时方式选择，该位为 1 时，选择 12 小时方式，该位为 0 时，选择 24 小时方式。在 12 小时方式下，时寄存器的第 5 位为 AM/PM 选择，该位为 1 时，选择 PM，该位为 0 时，选择 AM。在 24 小时方式下，时寄存器的第 5 位为第 2 个小时位（20～23）。

表 5-13　DS1302 内部寄存器地址与内容

寄存器名	命令字节		取值范围	寄存器内容							
	写	读		7	6	5	4	3	2	1	0
秒寄存器	80H	81H	00～59	CH	10s			SEC			
分寄存器	82H	83H	00～59	0	10min			MIN			
时寄存器	84H	85H	00～23 或 01～12	12/24	0	10A/P	HR	HR			
日期寄存器	86H	87H	01～28，29，30，31	0	0	10DATA		DATE			
月寄存器	88H	89H	01～12	0	0	0	10M	MONTH			
周寄存器	8AH	8BH	01～07	0	0	0	0	0	DAY		
年寄存器	8CH	8DH	00～99	10YEAR				YEAR			

例 5-5　图 5-14 所示为 LPC2138 与 DS1302 实现的万年历电路。DS1302 采用外部 3.3V 电源供电，LPC2138 的引脚 P1.28、P1.29、P1.30 分别连到 DS1302 的 RST、SCLK、I/O 端，LPC2138 的引脚 P1.16、P1.17、P1.18 分别连接“设置”、“增大”、“减小”按钮，用于对万年历进行设置和调整。万年历的显示部分采用 12864 点阵图型液晶显示模块，显示信息包括公历和农历。

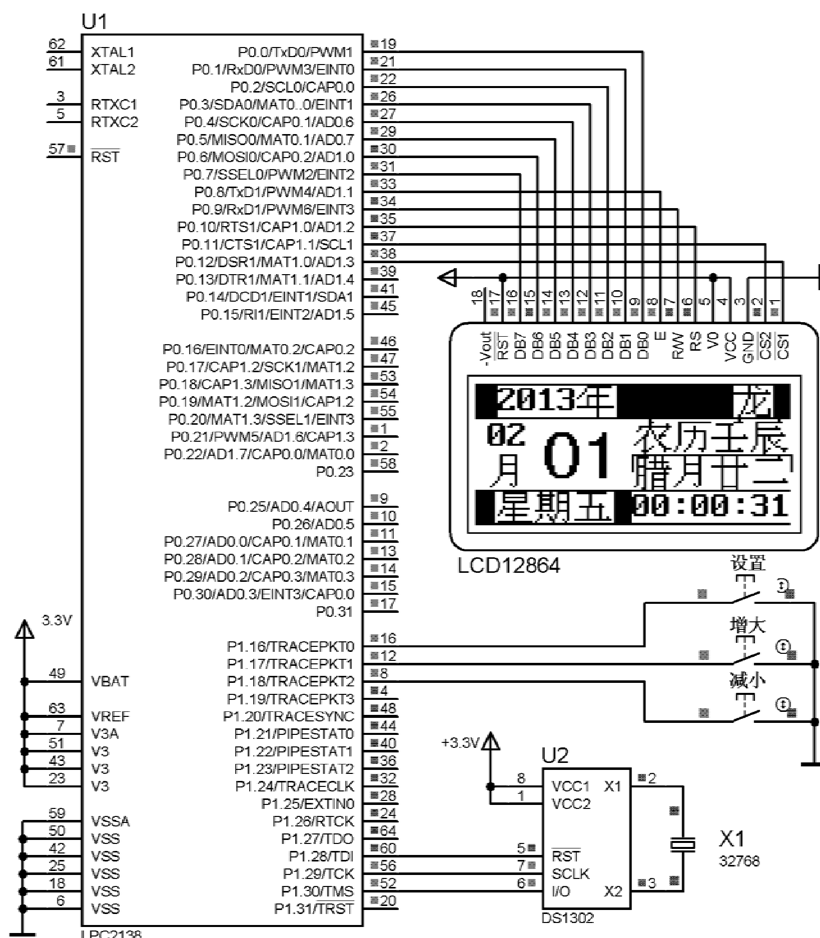


图 5-14　LPC2138 与 DS1302 实现的万年历电路

本例中 DS1302 驱动程序包括在 DS1302.C 模块中，具体如下：

```
#include   <iolpc2138.h>
#include "type.h"

#define     DS1302_RST          (1<<28)               // P1.28 实时时钟复位线引脚
#define     DS1302_SCLK         (1<<29)               // P1.29 实时时钟时钟线引脚
#define     DS1302_IO           (1<<30)               // P1.30 实时时钟数据线引脚
#define    IO_INCON   ((IO1DIR|(DS1302_SCLK|DS1302_RST))&(~(DS1302_IO)))     // 定义写端口
#define    IO_OUTCON   (IO1DIR|(DS1302_SCLK|DS1302_RST|DS1302_IO))           // 定义读端口
#define uint   unsigned int
#define SECOND 0x80        //秒
#define MINUTE 0x82        //分
#define HOUR1   0x84       //时
#define DAY      0x86      //天
#define MONTH1 0x88        //月
#define WEEK     0x8a      //星期
#define YEAR1    0x8c      //年

typedef struct systime{
      uint8        cYEAR1;
      uint8        cMon;
      uint8        cDay;
      uint8        cHOUR1;
      uint8        cMin;
      uint8        cSec;
      uint8        cWeek;
}SYSTIME;

/****************************************************************************
* 函数名称：DS1302_Init()
* 函数功能：DS1302 初始化
****************************************************************************/
void DS1302_Init(){
      PINSEL1=PINSEL1&(~(0x3f<<10));
      PINSEL2=PINSEL2&(~0x0C);
      IODIR1=(0xfff0<<15);
}

/****************************************************************************
* 函数名称：DS1302_Write()
* 函数功能：DS1302 写入
****************************************************************************/
void DS1302_Write(uint8 dat){
      uint8 i;
      for(i=8; i>0; i--){
            if((dat&0x01)!= 0)
```

```
			IO1SET = DS1302_IO;			//置高电平
		else
			IO1CLR = DS1302_IO;			//置低电平
		IO1SET = DS1302_SCLK;
		IO1CLR = DS1302_SCLK;
		dat >>= 1;					//数值右移一位，对原来的数值乘以 2
	}
}

/****************************************************************************
* 函数名称：DS1302_Read()
* 函数功能：DS1302 读出
* 出口参数：dat   读出的 DS1302 内容
****************************************************************************/
uint8 DS1302_Read(){
	uint8 i;
	uint8 dat=0;
	IO1DIR = IO_INCON;
	for(i=8; i>0; i--){
		dat >>= 1;
		if((IO1PIN&DS1302_IO) !=0 )
			dat += 0x80;
		IO1SET = DS1302_SCLK;
		IO1CLR = DS1302_SCLK;
	}
	IO1DIR = IO_OUTCON;
	return(dat);
}

/****************************************************************************
* 函数名称：WDS1302()
* 函数功能：DS1302 单字节写入
* 入口参数：ucAddr DS1302 寄存器地址，ucDat   写入的 DS1302 内容
****************************************************************************/
void WDS1302(uint8 ucAddr, uint8 ucDat){
	IO1CLR = DS1302_RST;
	IO1CLR = DS1302_SCLK;
	IO1SET = DS1302_RST;
	DS1302_Write(ucAddr);		// 写地址命令
	DS1302_Write(ucDat);		// 写 1B 数据
	IO1SET = DS1302_SCLK;
	IO1CLR = DS1302_RST;
}

/****************************************************************************
* 函数名称：RDS1302()
```

```
* 函数功能：DS1302 单字节读出
* 入口参数：ucAddr DS1302 寄存器地址
* 出口参数：ucDat   读出的 DS1302 内容
****************************************************************************/
uint8 RDS1302(uint8 ucAddr){
        uint8 ucDat;
        IO1CLR = DS1302_RST;
        IO1CLR = DS1302_SCLK;
        IO1SET = DS1302_RST;
        DS1302_Write(ucAddr);                    // 写地址命令
        ucDat=DS1302_Read();                     // 读取 DS1302 内容
        IO1SET = DS1302_SCLK;
        IO1CLR = DS1302_RST;
        return ucDat;
}

/****************************************************************************
* 函数名称：SetTime()
* 函数功能：时间设置
* 入口参数：SYSTIME sys  系统时间
****************************************************************************/
void SetTime(SYSTIME sys){
        WDS1302(YEAR1,sys.cYEAR1);
        WDS1302(MONTH1,sys.cMon&0x1f);
        WDS1302(DAY,sys.cDay&0x3f);
        WDS1302(HOUR1,sys.cHOUR1&0xbf);
        WDS1302(MINUTE,sys.cMin&0x7f);
        WDS1302(SECOND,sys.cSec&0x7f);
        WDS1302(WEEK,sys.cWeek&0x07);
}

/****************************************************************************
* 函数名称：GetTime()
* 函数功能：时间获取
* 入口参数：SYSTIME *sys  存放系统时间的数据指针
****************************************************************************/
void GetTime(SYSTIME *sys){
        uint8 uiTempDat;
        uiTempDat=RDS1302(YEAR1|0x01);
        (*sys).cYEAR1=(uiTempDat>>4)*10+(uiTempDat&0x0f);
        uiTempDat=RDS1302(0x88|0x01);
        (*sys).cMon=((uiTempDat&0x1f)>>4)*10+(uiTempDat&0x0f);
        uiTempDat=RDS1302(DAY|0x01);
        (*sys).cDay=((uiTempDat&0x3f)>>4)*10+(uiTempDat&0x0f);
        uiTempDat=RDS1302(HOUR1|0x01);
        (*sys).cHOUR1=((uiTempDat&0x3f)>>4)*10+(uiTempDat&0x0f);
```

```
        uiTempDat=RDS1302(MINUTE|0x01);
        sys->cMin=((uiTempDat&0x7f)>>4)*10+(uiTempDat&0x0f);
        uiTempDat=RDS1302(SECOND|0x01);
        sys->cSec=((uiTempDat&0x7f)>>4)*10+(uiTempDat&0x0f);
        uiTempDat=RDS1302(MONTH1|0x01);
        (*sys).cMon=uiTempDat&0x17;
        uiTempDat=RDS1302(WEEK|0x01);
        sys->cWeek=uiTempDat&0x07;
    }
```

进入主程序后首先调用 DS1302 初始化子程序进行引脚配置，接着进行日期初始化和 12864LCD 图形初始化，初始化完成之后，进入循环，在循环中完成获取 DS1302 当前时间、LCD 屏幕上显示万年历、按键扫描和时间设置等工作。

主程序文件 main.c 如下：

```
#include "Config.h"
#include "Target.h"
#include    <intrinsics.h>
#include "type.h"
#include "ds1302.h"
#include "lunar.h"
#include "model.h"
#include "12864.h"

#define NoUpLine       1
#define UpLine         0
#define NoUnderLine 1
#define UnderLine      0
#define FALSE   0
#define TRUE       1
#define KEY1          (1<<16)              // 定义 KEY1 引脚
#define KEY2          (1<<17)              // 定义 KEY2 引脚
#define KEY3          (1<<18)              // 定义 KEY3 引脚

uint8 Mon2[2][13]={0,31,28,31,30,31,30,31,31,30,31,30,31,
                   0,31,29,31,30,31,30,31,31,30,31,30,31};
uint8 Hour_Flag=TRUE,Min_Flag=TRUE,Sec_Flag=TRUE;      // 设置时间标志
uint8 Year_Flag=TRUE,Mon_Flag=TRUE,Day_Flag=TRUE;
uint8 State_Set=0;                                     // 设置时、分、秒、日、月和年等状态
uint8 State_Flag=FALSE,Inc_Flag=FALSE,Dec_Flag=FALSE;  // 按键是否按下的标志
uint8 dispBuf[7];
uint8 T0_Count=0,Tmp_Count=0;
SYSTIME sys;                                           // 系统日期
SPDATE SpDat;                                          // 农历日期

/************************************************************************
```

```
* 函数名称：LCD_ShowTime()
* 函数功能：LCD 显示
* 入口参数：
*           cDat:       要显示的数
*           X:          行数 0～7
*           Y:          列数 0～127
*           show_flag: 是否反白显示，0 反白，1 不反白
*           upline:     上划线, 0 表示带上划线
*           underline:  下划线, 0 表示带下划线
*************************************************************************/
void LCD_ShowTime(char cDat,uint8 X,uint8 Y,uint8 show_flag,uint8 up,uint8 under){
    uint8 s[2];
    s[0]=cDat/10+'0';
    s[1]=cDat%10+'0';
    en_disp(X,Y,2,Asc,s,show_flag,up,under);
}

/*************************************************************************
* 函数名称：Show_YMD()
* 函数功能：年、月、日、星期、农历、生肖显示
*************************************************************************/
void Show_YMD(){
    uint8 uiTempDat;
    uiTempDat=RDS1302(0x88|0x01);
    sys.cMon=((uiTempDat&0x1f)>>4)*10+(uiTempDat&0x0f);
    LCD_ShowTime(sys.cMon,2,5,Mon_Flag,NoUpLine,NoUnderLine);
    hz_disp(4,5,1,uMod[1],1,NoUpLine,NoUnderLine);                //月
    Show16X32(2,27,ucNum3216[sys.cDay/10],Day_Flag);              //日
    Show16X32(2,43,ucNum3216[sys.cDay%10],Day_Flag);
    hz_disp(6,8,2,ucLunar[13],1,UpLine,UnderLine);
    if(sys.cWeek==7)                                              //星期
        hz_disp(6,40,1,uMod[2],1,UpLine,UnderLine);
    else
        hz_disp(6,40,1,ucLunar[sys.cWeek],1,UpLine,UnderLine);
    LCD_ShowTime(20,0,9,1,UpLine,UnderLine);                      //年
    LCD_ShowTime(sys.cYear,0,25,Year_Flag,UpLine,UnderLine);
    hz_disp(0,41,1,uMod[0],1,UpLine,UnderLine);

    SpDat=GetSpringDay(sys.cYear,sys.cMon,sys.cDay);              //获得农历
    if(SpDat.cMon==1)                                             //显示农历月
        hz_disp(4,64,1,ucLunar[15],1,UpLine,NoUnderLine);         //"正"
    else if(SpDat.cMon==11)
        hz_disp(4,64,1,ucLunar[16],1,UpLine,NoUnderLine);         //"冬"
    else if(SpDat.cMon==12)
        hz_disp(4,64,1,ucLunar[17],1,UpLine,NoUnderLine);         //"腊"
    else
```

```
        hz_disp(4,63,1,ucLunar[SpDat.cMon],1,UpLine,NoUnderLine);       // "二"～"十"
    if(SpDat.cDay/10==1 && SpDat.cDay%10>0)        // 显示"十" 例如"十四"而不是"一四"
        hz_disp(4,95,1,ucLunar[10],1,UpLine,NoUnderLine);
    else if(SpDat.cDay/10==2 && SpDat.cDay%10>0)  // 显示"廿" 例如"廿四"而不是"二四"
        hz_disp(4,95,1,ucLunar[19],1,UpLine,NoUnderLine);
    else
        hz_disp(4,95,1,ucLunar[SpDat.cDay/10],1,UpLine,NoUnderLine);      // 正常数字
    if(!(SpDat.cDay%10))                                                    // "十"
        hz_disp(4,111,1,ucLunar[10],1,UpLine,NoUnderLine);
    else                                                                    // 正常数字
        hz_disp(4,111,1,ucLunar[SpDat.cDay%10],1,UpLine,NoUnderLine);

    hz_disp(0,104,1,SX[(uint16)(2000+SpDat.cYear)%12],1,UpLine,UnderLine);              // 生肖
    hz_disp(2,95,1,TianGan[(uint16)(2000+SpDat.cYear)%10],1,NoUpLine,NoUnderLine);  // 天干
    hz_disp(2,111,1,DiZhi[(uint16)(2000+SpDat.cYear)%12],1,NoUpLine,NoUnderLine);   // 地支
}

/**************************************************************************
* 函数名称：LCD_ShowWNL()
* 函数功能：万年历显示
***************************************************************************/
void LCD_ShowWNL(){
    LCD_ShowTime(sys.cSec,6,111,Sec_Flag,UpLine,UnderLine);  // 秒，每秒钟刷新
    if(!sys.cSec || State_Set)                                // 分，普通模式每分钟刷新
    LCD_ShowTime(sys.cMin,6,87,Min_Flag,UpLine,UnderLine);   // 设置模式每次刷新

    if(!sys.cSec && !sys.cMin || State_Set)                   // 时，普通模式每小时刷新
    LCD_ShowTime(sys.cHour,6,63,Hour_Flag,UpLine,UnderLine);// 设置模式每次刷新

    if(!sys.cSec && !sys.cMin && !sys.cHour || State_Set ){   // 公历农历的年、月、日、星期
        Show_YMD();                                           // 普通模式每天刷新
        if(State_Set==7) State_Set=0;                         // 设置模式每次刷新
    }
}

/**************************************************************************
* 函数名称：CAL_Init()
* 函数功能：日期初始化
***************************************************************************/
void CAL_Init(){
    sys.cYear=0x11;
    sys.cMon=0x10;
    sys.cDay=0x31;
    sys.cHour=0x23;
    sys.cMin=0x59;
    sys.cSec=0x55;
```

```
	sys.cWeek=GetWeekDay(sys.cYear,sys.cMon,sys.cDay);
}

/*****************************************************************************
* 函数名称：GUI_Init()
* 函数功能：LCD 图形初始化
*****************************************************************************/
void GUI_Init(){
	LCD12864_init();
	ClearLCD();
	Rect(0,0,127,63,1);                                          // 描绘框架
	Line(0,48,127,48,1);
	Line(0,15,127,15,1);
	Line(63,32,128,32,1);
	SetTime(sys);                                                // 设置时间
	GetTime(&sys);                                               // 获得时间
	Show_YMD();
	LCD_ShowTime(sys.cSec,6,111,Sec_Flag,UpLine,UnderLine);
	en_disp(6,103,1,Asc,":",1,UpLine,UnderLine);
	LCD_ShowTime(sys.cMin,6,87,Min_Flag,UpLine,UnderLine);
	en_disp(6,79,1,Asc,":",1,UpLine,UnderLine);
	LCD_ShowTime(sys.cHour,6,63,Hour_Flag,UpLine,UnderLine);
	hz_disp(2,64,1,ucLunar[11],1,NoUpLine,NoUnderLine);          // "农"
	hz_disp(2,80,1,ucLunar[12],1,NoUpLine,NoUnderLine);          // "历"
	hz_disp(4,79,1,uMod[1],1,UpLine,NoUnderLine);                // "月"
}

/*****************************************************************************
* 函数名称：DecToBCD()
* 函数功能：二-十进制转换
*****************************************************************************/
void DecToBCD(){
	sys.cHour=(((sys.cHour)/10)<<4)+((sys.cHour)%10);
	sys.cMin=(((sys.cMin)/10)<<4)+((sys.cMin)%10);
	sys.cSec=((sys.cSec/10)<<4)+((sys.cSec)%10);
	sys.cYear=((sys.cYear/10)<<4)+((sys.cYear)%10);
	sys.cMon=((sys.cMon/10)<<4)+((sys.cMon)%10);
	sys.cDay=((sys.cDay/10)<<4)+((sys.cDay)%10);
}

/*****************************************************************************
* 函数名称：Time_Set()
* 函数功能：时间设置
*****************************************************************************/
void Time_Set(){
	if(State_Flag){                                              // 设置键按下
```

```
        State_Flag=FALSE;
        State_Set++;
        if(State_Set==8) State_Set=0;
}
Hour_Flag=TRUE;Min_Flag=TRUE;Sec_Flag=TRUE;
Year_Flag=TRUE;Mon_Flag=TRUE;Day_Flag=TRUE;
switch(State_Set){                      // 设置类型
        case 0:                         // 无设置
                break;
        case 1:                         // 设置时
                Hour_Flag=FALSE;
                break;
        case 2:                         // 设置分
                Min_Flag=FALSE;
                break;
        case 3:                         // 设置秒
                Sec_Flag=FALSE;
                break;
        case 4:                         // 设置天
                Day_Flag=FALSE;
                break;
        case 5:
                Mon_Flag=FALSE;         // 设置月
                break;
        case 6:
                Year_Flag=FALSE;        // 设置年
                break;
        case 7:                         // 无动作，设置此值为让"年"的反白消失。
                break;
}

if(Inc_Flag){                           // 加键被按下
        Inc_Flag=FALSE;
        switch(State_Set){
                case 0:
                        break;
                case 1:                 // 小时加
                        sys.cHour++;
                        (sys.cHour)%=24;
                        break;
                case 2:                 // 分加 1
                        sys.cMin++;
                        sys.cMin%=60;
                        break;
                case 3:                 // 秒加 1
                        sys.cSec++;
```

```
				sys.cSec%=60;
				break;
			case 4:											// 天加 1
				(sys.cDay)=(sys.cDay%Mon2[YearFlag(sys.cYear)][sys.cMon])+1;
				break;
			case 5:											// 月加 1
				sys.cMon=(sys.cMon%12)+1;
				break;
			case 6:
				sys.cYear++;								// 年加 1
				sys.cYear=sys.cYear%100;
				break;
		}
		DecToBCD();											// 转为 BCD 数
		sys.cWeek=GetWeekDay(sys.cYear,sys.cMon,sys.cDay);	// 算出星期
		SetTime(sys);										// 存入 DS1302
	}

	if(Dec_Flag){											// 减键按下
		Dec_Flag=FALSE;
		switch(State_Set){
			case 0:
				break;
			case 1:
				sys.cHour=(sys.cHour+23)%24;				// 时减 1
				break;
			case 2:											// 分减 1
				sys.cMin=(sys.cMin+59)%60;
				break;
			case 3:											// 秒减 1
				sys.cSec=(sys.cSec+59)%60;
				break;
			case 4:											// 天减 1
				sys.cDay=((sys.cDay+Mon2[YearFlag(sys.cYear)][sys.cMon]-1)
								%Mon2[YearFlag(sys.cYear)][sys.cMon]);
				if(sys.cDay==0) sys.cDay=Mon2[YearFlag(sys.cYear)][sys.cMon];
				break;
			case 5:											// 月减 1
				sys.cMon=(sys.cMon+11)%12;
				if(sys.cMon==0) sys.cMon=12;
				break;
			case 6:											// 年减 1
				sys.cYear=(sys.cYear+99)%100;
				break;
		}
		DecToBCD();
```

```
            sys.cWeek=GetWeekDay(sys.cYear,sys.cMon,sys.cDay);
            SetTime(sys);
        }
}

/*****************************************************************************
* 函数名称：key_check()
* 函数功能：按键扫描
*****************************************************************************/
void key_check(){
        uint8 i;
        if((IOPIN1&(1<<16))==0){
            State_Flag=TRUE;i=1;
            while(i==1) {if((IOPIN1&(1<<16))!=0) i=0;}
        }
        if((IOPIN1&(1<<17))==0){
            Inc_Flag=TRUE; i=1;
            while(i==1) {if((IOPIN1&(1<<17))!=0) i=0;}
        }
        if((IOPIN1&(1<<18))==0){
            Dec_Flag=TRUE; i=1;
            while(i==1) {if((IOPIN1&(1<<18))!=0) i=0;}
        }
}

/*****************************************************************************
* 函数名称：main()
* 函数功能：主函数
*****************************************************************************/
int main(){
        DS1302_Init();
        CAL_Init();
        GUI_Init();
        while(1){
            GetTime(&sys);              // 获得时间
            LCD_ShowWNL();              // 显示万年历
            key_check();                // 扫描按键
            Time_Set();                 // 时间设置
        }
}
```

本例中还包括年历转换模块 lunar.c，负责完成公历与农历之间的转换。农历中采用天干地支纪年法，十天干与数字的对应关系如下：

甲、乙、丙、丁、戊、己、庚、辛、壬、癸

4、 5、 6、 7、 8、 9、 0、 1、 2、 3

十二地支与数字的对应关系如下：

子、丑、寅、卯、辰、巳、午、未、申、酉、戌、亥

4、 5、 6、 7、 8、 9、 10、 11、 0、1、 2、3

天干地支纪年法是天干在前，地支在后，代表天干的数字为公历年分的最后一位数字。计算地支时用年份数除以 12，后面的余数就代表某个地支，例如，2010 年最后一位数字是 0，对应的天干为庚，2010 除以 12，余数为 6，对应的地支为寅，所以 2010 年为庚寅年。限于篇幅这里不再列出年历转换模块 lunar.c 清单，读者可以从本书附带的光盘中找到该模块程序。

5.4 SD 卡与 LPC2138 的接口及应用

5.4.1 SD 卡简介

SD 卡是一种基于 NAND Flash 的存储卡，由于它具有安全性高、容量大、体积小、功耗低、非易失性等优点，在嵌入式系统中得到了广泛的应用。

SD 卡支持两种接口模式：SD 模式和 SPI 模式。SD 模式采用 4 根数据线并行传输，速度快，但协议实现复杂。SPI 模式以串行方式传输数据，速度性能与 SD 模式相比有所欠缺，但协议实现简单，操作方便。SD 卡作为 SPI 设备，SPI 通道由以下 4 个信号组成：CS（片选）、CLK（时钟）、DI（主机到 SD 卡的数据信号）和 DO（卡到主机的数据信号）。片选信号 CS 在整个 SPI 操作过程中，必须保持低电平有效；CLK 时钟用于同步；DI 不但传输数据，还发送命令；DO 除了传输数据外，还发送应答信号。本节主要介绍 SD 卡在 SPI 模式下与 LPC2138 的接口及应用。

图 5-15 所示为 SD 卡的引脚排列，共有 9 个引脚，SD 模式下与 SPI 模式下各引脚的定义不相同。在 SPI 模式下，SD 卡 1～7 引脚依次为片选引脚 CS、数据输入 DI、地、电源 VDD、时钟信号 CLK、地、数据输出 DO，8 脚与 9 脚在 SPI 模式下保留未用。

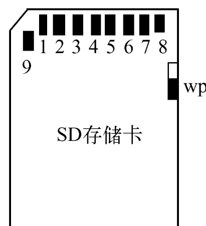


图 5-15 SD 卡引脚排列

上电后 SD 卡默认工作于 SD 模式，切换到 SPI 模式的方法是，上电后保持片选信号 CS 为低电平并延时大于 74 个 CLK 时钟周期的时间，然后发送复位命令 CMD0，复位成功（当接收到 0x01 的响应）后，再连续发送 CMD55 和 ACMD41，直到接收到响应 0x00 为止，此时 SD 卡已经进入 SPI 模式。上述所说的 CMD55 和 ACMD41 都是 SD 卡特有的命令。SD 卡命令由 6 个字节共 48 位组成，其格式如表 5-14 所列。

表 5-14 SD 卡命令格式

位标识	47	46	45:40	39:8	7:1	0
宽度	1	1	6	32	7	1
取值	0	1	×	×	×	1
描述	起始位	传输位	命令序号	命令参数	CRC7	停止位

应用时可以按以下顺序连续发送 6 个字节来实现上述 SD 卡命令格式。

字节 1：0 1 x x x x x x(命令号，由指令标志定义，如 CMD39 为 00100111，即十六进制 0x27，那么完整的 CMD39 第一字节为 01100111，即 0x27+0x40)。

字节 2～5:Command Arguments（命令参数，有些命令没有参数）。

字节 6：前 7 位为 CRC（循环冗余校验位），最后一位为停止位 1。

SD 卡命令共分为 12 类，分别为 Class0～Class11。不同的 SD 卡，主机根据其功能，支持不同的命令集。

① Class0（卡识别、初始化等基本命令集）：

CMD0：复位 SD 卡。

CMD1：读 OCR 寄存器。

CMD9：读 CSD 寄存器。

CMD10：读 CID 寄存器。

CMD12：停止读多块时的数据传输。

CMD13：读 Card_Status 寄存器。

② Class2（读卡命令集）：

CMD16：设置块的长度。

CMD17：读单块。

CMD18：读多块，直至主机发送 CMD12 为止。

③ Class4（写卡命令集）：

CMD24：写单块。

CMD25：写多块。

CMD27：写 CSD 寄存器。

④ Class5（擦除卡命令集）：

CMD32：设置擦除块的起始地址。

CMD33：设置擦除块的终止地址。

CMD38：擦除所选择的块。

⑤ Class6（写保护命令集）：

CMD28：设置写保护块的地址。

CMD29：擦除写保护块的地址。

CMD30：查询 SD 卡写保护位的状态。

⑥ Class7（卡的锁定，解锁功能命令集）。

⑦ Class8（申请特定命令集）。

⑧ Class10～11（保留）。

其中 Class1、Class3、Class9 不支持 SPI 模式。

SD 卡的所有操作都必须由命令完成，通过向 SD 卡发送相关命令并读取相应的响应来实现对 SD 卡的控制。在对 SD 卡进行读写之前，先要进行初始化操作，这是确保 SD 卡能在 SPI 模式下进行正常数据读写的前提。SD 卡的基本读写操作命令有：数据块读命令

（CMD17）、多数据块读命令（CMD18）、数据块写命令（CMD24）、多数据块写命令（CMD25）等。需要注意的是，在发送使 SD 卡空闲命令（CMD0）之前，应至少等待 74 个 CLK 时钟周期，确保 SD 卡进入 SPI 模式。初始化完成之后，如果采用默认的块读写长度 512B，则可以直接进行 SD 卡的读写；也可以采用 CMD16 命令来设置 SD 卡的块读取长度，可以是 1～512B 之间的任意值，但是对 SD 卡的写过程则要求快，长度必须为 512B。无论读写操作，都要求在读写命令发出后，有 1 个字节的数据起始令牌 FEH，数据传输结束后，有 2 个字节的 CRC 循环冗余校验编码。

完成 SD 卡的底层驱动之后，就可以按照 FAT 文件系统，对 SD 卡上的数据进行操作，进而在 SD 卡上创建文件、读写文件、删除文件等，从而实现文件的管理。

5.4.2 FAT16 文件系统

FAT 是由微软发布的一种磁盘文件管理系统。在 FAT 发展过程中，先后发布了 FAT12、FAT16、FAT32 三个版本，其中 FAT16 是指磁盘的一个分区最多含有 2 的 16 次方个簇，由于每个簇的最大存储空间只有 32KB，因此磁盘一个分区的存储容量最大为 2GB。对于容量大小不超过 2GB 的 SD 卡，通常只作为一个分区。

FAT16 主要由表 5-15 所列的 6 部分组成（主要是前 5 部分）。FAT16 在引导扇区（DBR）之后没有留任何保留扇区，其后紧随的便是 FAT 表。FAT 表是用来记录磁盘数据区簇链结构的，FAT 将磁盘空间按一定数目的扇区为单位进行划分，这样的单位称为簇。通常情况下，每扇区 512B。簇的大小一般是 2n（n 为整数）个扇区的大小，像 512B，1KB，2KB，4KB，8KB，16KB，32KB，64KB 等。实际中通常不超过 32KB。之所以要以簇为单位而不以扇区为单位进行磁盘的分配，是因为当分区容量较大时，采用大小为 512B 的扇区管理会增加 FAT 表的项数，对大文件存取增加消耗，文件系统效率不高。

表 5-15　FAT16 的主要组成

引导扇区 DBR （1 扇区）	FAT 表 1 （取实际大小）	FAT 表 2 （同 FAT1）	根文件夹及所有文件 （32 个扇区）	本文件夹及所有文件 （开始簇编号）	剩余扇区数 （不足一簇）

FAT 文件系统的目录结构如同一棵有向的从根到叶的树，这里提到的有向是指对于 FAT 分区内的任一文件（包括文件夹），均需从根目录寻址来找到。可以认为目录存储结构的入口就是根目录。

FAT 文件系统根据根目录来寻址其他文件（包括文件夹），故而根目录的位置必须在磁盘存取数据之前得以确定。FAT 文件系统就是根据分区的相关 DBR 参数与 DBR 中存放的已经计算好的 FAT 表的大小来确定的。磁盘格式化之后，根目录的大小和位置就都已经确定下，其位置紧随 FAT2 之后，大小通常为 32 个扇区。根目录之后便是存放文件的数据区。

FAT 文件系统的一个重要思想是把目录（文件夹）当作一个特殊的文件来处理，在 FAT16 中，虽然根目录地位并不等同于普通的文件，但其组织形式和普通的目录（文件夹）并没有不同。FAT 分区中所有的文件夹（目录）文件，实际上可以看作是一个存放其他文件（文件夹）入口参数的数据表。所以目录占用空间的大小并不等同于其下所有数据的大小，

但也不等于 0。通常是占一个很小的空间，可以看作目录文件是一个简单的二维表文件。其具体存储原理是：不管目录文件所占空间为多少簇，一簇为多少字节，系统都会以 32 个字节为单位进行目录文件所占簇的分配。以这 32 个字节确定的偏移，来定义本目录下的一个文件（或文件夹）的属性。这 32 个字节中各字节偏移定义如表 5-16 所列。

表 5-16　FAT16 目录项 32 个字节的表示定义

<table>
<tr><th>字节偏移(16 进制)</th><th>字节数</th><th colspan="2">定义</th></tr>
<tr><td>0x0～0x7</td><td>8</td><td colspan="2">文件名</td></tr>
<tr><td>0x8～0xA</td><td>3</td><td colspan="2">扩展名</td></tr>
<tr><td rowspan="7">0xB</td><td rowspan="7">1</td><td rowspan="7">属性字节</td><td>00000000(读写)</td></tr>
<tr><td>00000001(只读)</td></tr>
<tr><td>00000010(隐藏)</td></tr>
<tr><td>00000100(系统)</td></tr>
<tr><td>00001000(卷标)</td></tr>
<tr><td>00010000(子目录)</td></tr>
<tr><td>00100000(归档)</td></tr>
<tr><td>0xC～0x15</td><td>10</td><td colspan="2">系统保留</td></tr>
<tr><td>0x16～0x17</td><td>2</td><td colspan="2">文件的最近修改时间</td></tr>
<tr><td>0x18～0x19</td><td>2</td><td colspan="2">文件的最近修改日期</td></tr>
<tr><td>0x1A～0x1B</td><td>2</td><td colspan="2">表示文件的首簇号</td></tr>
<tr><td>0x1C～0x1F</td><td>4</td><td colspan="2">表示文件的长度</td></tr>
</table>

下面对表 5-16 中的一些取值进行说明：

- 对于短文件名，系统将文件名分成两部分进行存储，即主文件名+扩展名。0x0～0x7 字节记录文件的主文件名，0x8～0xA 记录文件的扩展名，取文件名中的 ASCII 码值。不记录主文件名与扩展名之间的“.”号。主文件名不足 8 个字符以空白符（20H）填充，扩展名不足 3 个字符同样以空白符(20H)填充。0x00 偏移处的取值若为 00H，表明目录项为空；若为 E5H，表明目录项曾被使用，但对应的文件或文件夹已被删除。（这也是误删除后恢复的理论依据）。文件名中的第一个字符若为“.”或“..”表示这个簇记录的是一个子目录的目录项。“.”代表当前目录；“..”代表上级目录。
- 0xB 字节为属性字段，系统将 0xB 的一个字节分成 8 位，用其中的一位代表某种属性的有或无。例如 00000101 就表示个文件的属性是只读、系统。
- 0xC～0x15 字节在 FAT16 的定义中为保留未用，在高版本的 WINDOWS 系统中有时也用它来记录修改时间和最近访问时间。
- 0x16～0x17 字节为文件的最近修改时间。0x16 字节的 0～4 位是以 2s 为单位的量值；0x16 字节的 5～7 位和 0x17 字节的 0～2 位是分钟；0x17 字节的 3～7 位是小时。
- 0x18～0x19 字节为文件的最近修改日期。0x18 字节的 0～4 位是日期数；0x18 字节 5～7 位和 0x19 字节 0 位是月份；0x19 字节的 1～7 位为年号。

- 0x1A～0x1B 字节存放文件或目录的首簇号。系统根据首簇号在 FAT 表中找到入口，然后再跟踪簇链直至簇尾，同时用 0x1C～0x1F 字节所表示的文件长度来判定文件有效性，就可以正确地读取文件（目录）了。
- 普通子目录的寻址过程也是通过其父目录中的目录项来指定的，与数据文件（指非目录文件）不同的是，目录项偏移 0xB 的第 4 位置 1，而数据文件为 0。

读取 SD 卡中的文件时，首先要根据文件名查找该文件的信息结构体。根据文件信息结构体中的起始簇号即可找到数据区第 1 簇的内容，也可以在 FAT 表中找到第 2 个簇号，根据第 2 个簇号又能找到第 2 个簇的内容和 FAT 表中的第 3 个簇号，直至遇到文件结束标志。这样就可以根据 FAT 表中的簇号读出全部文件数据。下面以图 5-16 为例说明读取 SD 卡文件的具体步骤。

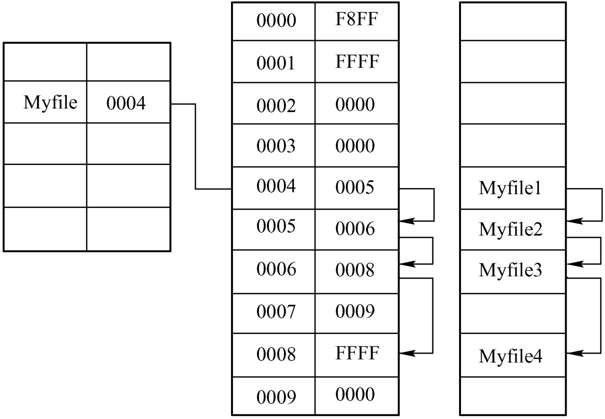


图 5-16　SD 卡文件的读取步骤

1）先在目录项表中找到与文件名匹配的内容，如 Myfile，从 Myfile 所对应的目录项中可以找到该文件的首簇号 0004。

2）根据首簇号 0004 访问 FAT 表，读出首簇号对应的 FAT 表项内容 0005，即第 2 个簇号。根据第 2 个簇号再访问 FAT 表，读出其对应的内容，即第 3 个簇号 0006，这样依次做下去，直到最后一个表项内容为 FFFF 为止。

3）由第 2 步可知，Myfile 这个文件占用了 4 个簇，这 4 个簇号形成一个簇链：0004-0005-0006-0008，根据这些簇号所形成的簇链访问者 4 个簇号对应的 4 个数据存储区，文件 Myfile 就分别存放在这 4 个存储区中。

5.4.3　SD 卡接口应用举例

例 5-6　图 5-17 所示为 LPC2138 与 SD 卡的接口应用电路。SD 卡以 SPI 模式工作，为了提高工作速度，采用 LPC2138 的 SPI1，即 SSP 接口功能，电路中还扩展了 T6963C 点阵图形 LCD 模块，用于显示读写 SD 卡的数据；通过 UART0 连接虚拟终端，实现人机交互完成各种操作。另外 Proteus 提供的 SD 卡模型中，只能够识别扩展名为.mmc 的 SD 卡文件，该文件需要通过专用软件 WinHex 或 WinImage 来创建。

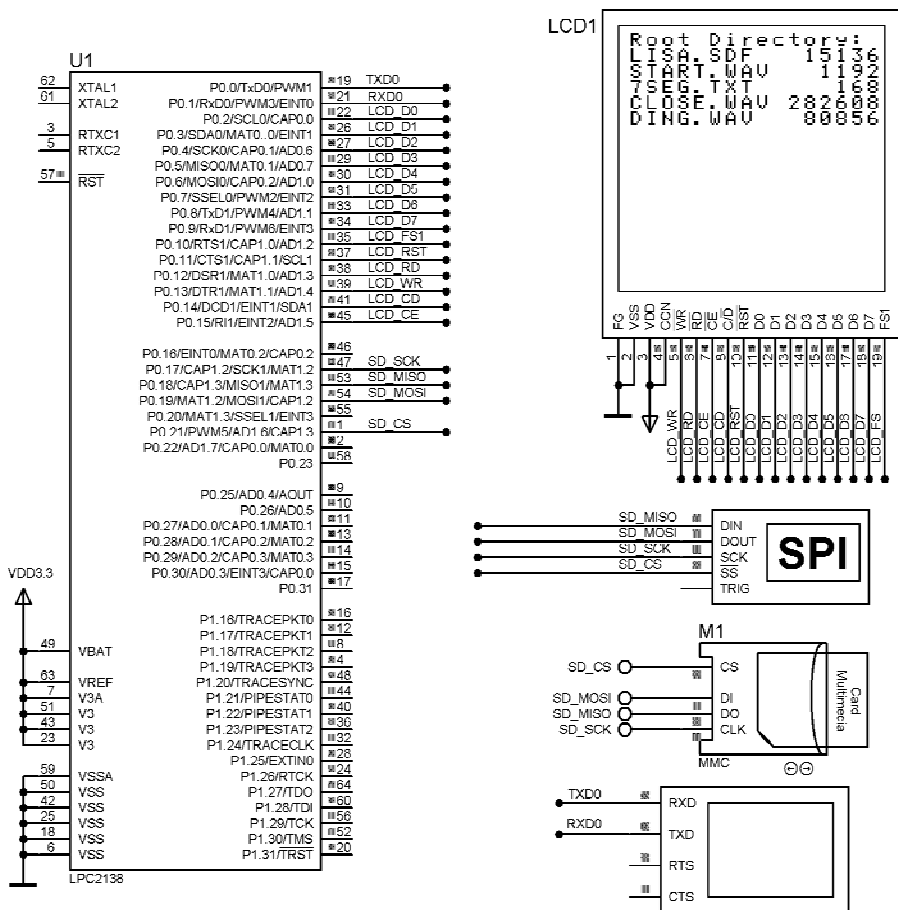


图 5-17　LPC2138 与 SD 卡的接口应用电路

本例中 LPC2138 的 SSP 驱动程序包括在 LPC2138SPI.C 模块中，具体如下：

```
#include "LPC2138SPI.h"
/*****************************************************************************
* 名称：SPI0_Init()
* 功能：SPI 接口初始化，设置为主机模式
*****************************************************************************/
void SPI0_Init(void){
     S0SPCCR = 0x08;   // 设置 SPI 时钟分频，SPI 时钟＝PCLK/SPCCR，SPCCR 必须大于或等于 8
     S0SPCR = 0x20;    // MSTR=1，主模式选择
}

/*****************************************************************************
* 名称：SPI0_SendData()
* 功能：向 SPI 总线发送数据或者读取数据。
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
*****************************************************************************/
```

```
uint8 SPI0_SendData(uint8 data){
        S0SPDR = data;
        while( 0==(S0SPSR&0x80));
        return(S0SPDR);
}

/*****************************************************************************
** 函数名称 ：SSP_Init()
** 函数功能 ：将 SSP 控制器初始化 SPI 接口，设置为主机
*****************************************************************************/
void   SSP_Init(void){
      SSPCR0 = (0x01 << 8) |        // SCR 设置 SPI 时钟分频
               (0x00 << 7) |        // CPHA 时钟输出相位，仅 SPI 模式有效
               (0x00 << 6) |        // CPOL 时钟输出极性，仅 SPI 模式有效
               (0x00 << 4) |        // FRF 帧格式 00=SPI，01=SSI，10=Microwire，11=保留
               (0x07 << 0);         // DSS 数据长度，0000-0010=保留，0011=4 位，0111=8 位，1111=16 位

      SSPCR1 = (0x00 << 3) |        // SOD   从机输出禁能，1=禁止，0=允许
               (0x00 << 2) |        // MS    主从选择，0=主机，1=从机
               (0x01 << 1) |        // SSE   SSP 使能，1=允许 SSP 与其他设备通信
               (0x00 << 0);         // LBM   回写模式

      SSPCPSR = 0x06;               // PCLK 分频值
      SSPIMSC = 0x07;               // 中断屏蔽寄存器
      SSPICR   = 0x03;              // 中断清除寄存器
}

/*****************************************************************************
* 函数名称：SSP_SendData()
* 函数功能：SSP 接口向 SPI 总线发送数据
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
*****************************************************************************/
uint8   SSP_SendData(uint8 data){
        uint8 i;
        SSPDR = data;
        for(i=0; i<12; i++);
        while( (SSPSR & 0x10) );
        return(SSPDR);
}

/*****************************************************************************
* 函数名称：SPI_SendData()
* 函数功能：根据定义采用 SSP 或 SPI 接口向 SPI 总线发送数据
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
```

```
******************************************************************************/
uint8 SPI_SendData(uint8 data){
#ifdef SPI_USING_SSP
        return SSP_SendData(data);
#else
        return SPI0_SendData(data);
#endif
}
```

本例中 SD 卡驱动程序包括在 SD.C 模块中，列表如下：

```
#include "LPC2138SPI.h"
#include "SD.h"

#define SD_Disable()              IO0SET = SD_CS;
#define SD_Enable()               IO0CLR = SD_CS;
#define SD_CS_DIR_IN()            IO0DIR &=  ～SD_CS;
#define SD_CS_DIR_OUT()           IO0DIR |= SD_CS;
#define SD_CS_DATA_IN()           (IO0PIN & SD_CS)

extern int temp1;

/******************************************************************************
* 函数名称：SD_SendData()
* 函数功能：根据定义采用 GPIO 或 SPI 接口向 SPI 总线发送数据
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
******************************************************************************/
uint8 SD_SendData(uint8 data){
#ifdef SD_USING_GPIO
   return SD_Write(data);
#else
   return SPI_SendData(data);
#endif
}

/******************************************************************************
* 函数名称：SD_DetectCard()
* 函数功能：检测 SD 卡是否存在
* 出口参数：1 为 SD 卡存在，0 为 SD 卡不存在
******************************************************************************/
boolean SD_DetectCard(void){
      SD_CS_DIR_IN();
      delay(18);
      if((SD_CS_DATA_IN()) == 0){
            SD_CS_DIR_OUT();
            return FALSE;
      }
```

```
    SD_CS_DIR_OUT();
    return TRUE;
}

/*******************************************************************************
* 函数名称：SD_WriteCommand()
* 函数功能：向 SD 卡发送命令
* 入口参数：command 命令，argument 参数，CRC 冗余校验码
* 出口参数：tmp   SD 卡响应值
*******************************************************************************/
uint8 SD_WriteCommand(uint8 command, uint32 argument, uint8 CRC){
     uint8 tmp = 0xFF;
     uint16 Timeout = 0;
     SD_Disable();                  //片选信号拉高
     SD_SendData(0xFF);             //发送 8 个时钟
     SD_Enable();                   //片选信号拉低
     SD_SendData(command|0x40);
     SD_SendData(argument);
     SD_SendData(argument>>8);
     SD_SendData(argument>>16);
     SD_SendData(argument>>24);
     SD_SendData(CRC);
     do{
          tmp = SD_SendData(0xFF);
          if (Timeout++ > 500){
               break;               //响应超时
          }
     }while (tmp == 0xFF);
     return tmp;
}

/*******************************************************************************
* 函数名称：SD_CheckWriteState()
* 函数功能：检测前面的写操作是否成功
* 出口参数：1 为写入成功，0 为写入失败
*******************************************************************************/
boolean SD_CheckWriteState(void){
    unsigned int count = 0;
    char inRes;
    while(count <= 64){
        inRes = SD_SendData(0xFF);
        if((inRes & 0x1F) == 0x05)
            break;
        else if((inRes & 0x1F) == 0x0B)
            return FALSE;
        else if((inRes & 0x1F) == 0x0D)
```

```
                return FALSE;
            count++;
        }
        count = 0;
        while((count < 64) && (SD_SendData(0xFF) == 0x00));
        if(count < 64)
            return TRUE;
        else
            return FALSE;
}

/***************************************************************************
* 函数名称：SD_Init()
* 函数功能：SD 卡初始化
* 入口参数：无
* 出口参数：0 为初始化成功，其他值为初始化失败
***************************************************************************/
uint8 SD_Init(void){
        uint8 i;
        uint16 Timeout = 0;
#ifdef SPI_USING_SSP
        PINSEL1 |= 0x000000A8;                  // 设置引脚连接到 SSP
        IO0DIR |= SD_CS;
        SSP_Init();
#else
        PINSEL0 |= 0x1500;                      // 设置引脚连接到 SPI0
        IO0DIR |= SD_CS;
        SPI0_Init();
#endif
        SD_Disable();                           // 禁止 SD 卡

/*  等待 SD 卡上电启动完成，等待过程中发送的参数须为 0xFF 才能确保 SD 卡正确启动  */
        for(i=0;i<=0x10;i++)
        SPI_SendData(0xFF);
        SD_Enable();                            // 使能 SD 卡

/*  发送 SD 命令 GO_IDLE_STATE(0x40)，使 SD 卡进入空闲状态，其他 4 个参数都为 0，
    最后一个参数是 CRC 校验码，用手工的方法计算出来，在进入 SPI 模式后，CRC 校验不再需要。
    初始化过程中片选信号的有效状态（低电平有效），将使 SD 卡进入 SPI 模式
*/
        while(SD_WriteCommand(GO_IDLE_STATE,0,0x95) != 0x01){
                if (Timeout++ > 5000){          // 等待 SD 卡响应，对于初始化命令响应值应为 0x01
                        printf("timeout code: 1\r\n");
                        return(1);              //响应超时
                }
        }
```

```
        Timeout = 0;
/*  发送 SD 卡初始化命令 CMD55+ACMD41 */
#if 0
      do{
          r1 = SD_WriteCommand(APP_CMD, 0, 0);
          printf("r1=%d\r\n", r1);
          if (r1 != 0x01){
                printf("cmd55 failed %x\n\r", r1);
                return;
          }
          r1 = SD_WriteCommand(41, 0x40000000, 0);
          printf("r1=%d\r\n", r1);
          if (Timeout++ > 200){
                printf("cmd41 failed %x\n\r", r1);
                return;                          //超时则返回 r1 状态
          }
      } while (r1 != 0);
#else
/*  发送指令 OUT_IDLE_STATE(0x41)，使 SD 卡退出空闲状态  */
while( SD_WriteCommand(OUT_IDLE_STATE, 0x00000000, 0xFF) != 0){
        if (Timeout++ > 5000){
              printf("timeout code: 2\r\n");
              return(2);                         //响应超时
        }
}
#endif
        SD_Disable();                            // 禁止 SD 卡
   /*  空操作，提供 8 个额外的时钟周期，让 SD 卡可以完成内部操作  */
        SPI_SendData(0xFF);
        printf("SD Initialize sucessfully!\r\n");
        return(0);
}

/*****************************************************************************
*  函数名称：SD_ReadBlock()
*  函数功能：SD 卡块读取
*  入口参数：sector  扇区，*Buffer  缓冲区指针
*  出口参数：BlockSize  块长度
*****************************************************************************/
uint16 SD_ReadSector(uint32 sector,uint8 *Buffer){
        uint8 tmp;
        uint16 i;
        uint32 arg=0;
        sector = sector << 9;                    //addr = addr * 512
        arg = ((sector & 0xFF000000) >> 24)
                |((sector & 0x00FF0000) >> 8)
```

```
            |((sector & 0x0000FF00) << 8);
        if (SD_WriteCommand (READ_SINGLE_BLOCK, arg, 0x00) != 0){
            return 0;
        }
        do{
                tmp = SD_SendData(0xFF);
        }while (tmp != SD_STARTBLOCK_READ);

        for (i=0; i<BlockSize; i++){
                *Buffer++ = SD_SendData(0xFF);
        }
SD_SendData(0xFF);                          //CRC
SD_SendData(0xFF);                          //CRC
SD_Disable();
return BlockSize;
}

/*************************************************************************
* 函数名称：SD_WriteBlock()
* 函数功能：SD 卡块写入
* 入口参数：sector 扇区，*Buffer 缓冲区指针
* 出口参数：BlockSize 块长度
*************************************************************************/
uint16 SD_WriteSector(uint32 sector,uint8 *Buffer){
        uint32 arg=0;
        unsigned int count = 0;
        sector = sector << 9;               //addr = addr * 512
        arg = ((sector & 0xFF000000) >> 24)
              |((sector & 0x00FF0000) >> 8)
              |((sector & 0x0000FF00) << 8);
        if (SD_WriteCommand (WRITE_SINGLE_BLOCK, arg, 0xFF) == 0){
              SD_SendData(0xFF);
              SD_SendData(0xFE);
              for(count = 0; count < SECTOR_SIZE; count++){
                      SD_SendData(Buffer[count]);
               }
               SD_SendData(0xFF);
               SD_SendData(0xFF);
               SD_CheckWriteState();
        }
        SD_Disable();
        SD_SendData(0xFF);
        return BlockSize;
}

/*************************************************************************
```

```
* 函数名称：SD_SetBlockLength()
* 函数功能：把 SD 卡的块长度初始化为 512B
* 出口参数：1 为设置块长度成功，0 为设置块长度失败
****************************************************************************/
boolean SD_SetBlockLength(void){
#ifdef ON_VSM_SIMULATION          //Proteus 的 SD 卡模型不支持此命令
	return TRUE;
#else
	SD_Enable();                  // 使能 SD 卡

/* 发送命令 SET_BLOCK_LEN，设置存储卡使用的块长度 */
	if(SD_WriteCommand(SET_BLOCK_LEN,0x0000200,0xFF) != 0){
		SPI_SendData(0xFF);       // 指令操作成功  释放总线
		SD_Disable();
		return TRUE;
	}
	SPI_SendData(0xFF);           // 释放总线
	SD_Disable();                 // 禁止 SD 卡
	return FALSE;
#endif
}

/****************************************************************************
* 函数名称：readPartialMultiSector()
* 函数功能：多扇区读取，并将读取的内容存入缓冲区
* 入口参数：sector1 读取的第 1 个扇区，sector2 读取的第 2 个扇区，
		*buf  缓冲区指针，bytesOffset 读取扇区的起始偏移量，
		bytesToRead  要读取的字节数
* 出口参数：countRead  读取的字节数
****************************************************************************/
unsigned int readPartialMultiSector(unsigned long sector1,
                                    unsigned long sector2,
                                    unsigned int byteOffset,
                                    unsigned int bytesToRead,
                                    unsigned char *buf){
	int count = 0;
	int countRead = 0;
	uint32 arg=0;
	uint8 tmp;
	sector1 = sector1 << 9;
	sector1 += 0;
	sector2 = sector2 << 9;
	sector2 += 0;
	SD_Enable();                          // 使能 MMC 卡
	arg = ((sector1 & 0xFF000000) >> 24)
	   |((sector1 & 0x00FF0000) >> 8)
```

```
        |((sector1 & 0x0000FF00) << 8);
    if(SD_WriteCommand(READ_SINGLE_BLOCK, arg, 0xFF) == 0x00){
        do{
          tmp = SD_SendData(0xFF);
        }while (tmp != SD_STARTBLOCK_READ);
        for(count = 0; count < byteOffset; count++)
            SD_SendData(0xFF);
        for(count = byteOffset; count < SECTOR_SIZE; count++, countRead++){
            buf[countRead] = SD_SendData(0xFF);
        }
        SD_SendData(0xFF);
        SD_SendData(0xFF);
    }
    else{
        return 0;
    }
    SD_Disable();
    SD_SendData(0xFF);
    SD_SendData(0xFF);
    SD_Enable();
    arg = ((sector2 & 0xFF000000) >> 24)
          |((sector2 & 0x00FF0000) >> 8)
          |((sector2 & 0x0000FF00) << 8);
    if(SD_WriteCommand(READ_SINGLE_BLOCK, arg, 0xFF) == 0x00){
        do{
          tmp = SD_SendData(0xFF);
        }while (tmp != SD_STARTBLOCK_READ);
        for(count = 0; countRead < bytesToRead; count++, countRead++){
            buf[countRead] = SD_SendData(0xFF);
        }
        for(count = count; count < SECTOR_SIZE; count++)
            SD_SendData(0xFF);
            SD_SendData(0xFF);
            SD_SendData(0xFF);
    }
    else{
        return 0;
    }
    SD_Disable();
    SD_SendData(0xFF);
    return countRead;
}

/*****************************************************************************
* 函数名称：readPartialSector()
* 函数功能：扇区读取，并将读取的内容存入缓冲区
```

```
* 入口参数：sector 读取扇区，*buf  缓冲区指针，bytesOffset 读取扇区的起始偏移量，
            bytesToRead  要读取的字节数
* 出口参数：countRead  读取的字节数
************************************************************************/
unsigned int readPartialSector(unsigned long sector,
                               unsigned int byteOffset,
                               unsigned int bytesToRead,
                               unsigned char *buf){
    int count = 0;
    int countRead = 0;
    int leftover = SECTOR_SIZE - byteOffset - bytesToRead;
    uint32 arg=0;
    uint8 tmp;

    temp1 = sector;
    sector += 0;
        SD_Enable();
        sector = sector << 9;
        arg = ((sector & 0xFF000000) >> 24)
            |((sector & 0x00FF0000) >> 8)
            |((sector & 0x0000FF00) << 8);
        if(SD_WriteCommand(READ_SINGLE_BLOCK, arg, 0xFF) == 0x00){
           do{
              tmp = SD_SendData(0xFF);
           }while (tmp != SD_STARTBLOCK_READ);
              for(count = 0; count < byteOffset; count++)
                  SD_SendData(0xFF);
              for(countRead = 0; countRead < bytesToRead; countRead++){
                  buf[countRead] = SD_SendData(0xFF);
              }
              for(count = 0; count < leftover; count++)
                 SD_SendData(0xFF);
                 SD_SendData(0xFF);
                 SD_SendData(0xFF);
        }
        SD_Disable();
        SD_SendData(0xFF);
        return countRead;
}
```

进入主程序后首先进行引脚配置，接着 T8983C 点阵图形 LCD 初始化、UART0 初始化和 FAT 文件系统初始化，初始化完成之后进入循环，在循环中，完成对各种输入命令的处理，实现对 SD 卡上文件的读写以及通过 UART0 和 LCD 屏幕显示操作结果。

主程序文件 main.c 列表如下：

```
#include "Config.h"
#include "Target.h"
#include <intrinsics.h>
#include "LPC2138SPI.h"
#include "T6963Driver.h"
#include "SD.h"
#include "FATLib.h"
#include "LPCUart.h"
#include "wav.h"

char const    SEND_STRING[] = "Hello from ARM7 LPC2138 System!\r\n";
uint8 buffer[512];
int temp1, temp2;
extern wav_param pwav;

/****************************************************************************
*  函数名称：debugPrintHexTable()
*  函数功能：以 HEX 表格形式输出存储器内容
*  入口参数：length    数据长度，   *buffer    缓冲区指针
****************************************************************************/
void debugPrintHexTable(uint16 length, uint8 *buffer){
    uint8 i;
    uint16 j;
    uint8 *buf;
    uint8 s;
    buf = buffer;
    /*  输出表头  */
    printf("     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F    0123456789ABCDEF\r\n");
    printf("         -----------------------------------------------    ---- ASCII -----\r\n");
    /*  输出数据  */
    for(j=0; j<((length+15)>>4); j++){
        printf("%04X",j<<4);
        printf("-");
        for(i=0; i<0x10; i++){
            if( ((j<<4)+i) < length){
                printf("%02X",buf[(j<<4)+i]);
                printf(" ");
            }
            else{
                printf("      ");
            }
        }
        /*  输出 ascii 数据  */
```

```
        for(i=0; i<0x10; i++){
            if( ((j<<4)+i) < length){
                s = buf[(j<<4)+i];
                if(s >= 0x20){
                    printf("%c",s);
                }
                else{
                    printf(".");
                }
            }
            else {
            }
        }
        printf("\r\n");
    }
}

/*******************************************************************************
* 函数名称：printHelpMsg()
* 函数功能：输出帮助信息
*******************************************************************************/
void printHelpMsg(){
    printf("\r\n");
    printf("\r\n");
    printf("Supported Command:\r\n\r\n");
    printf(" help -- show this command\r\n");
    printf(" del -- delete a file\r\n");
    printf(" dir    -- list the file in the root directory.\r\n");
    printf(" fs -- show the file system information.\r\n");
    printf(" ls    -- list the file in the root directory.\r\n");
    printf(" open -- open a file, and print the content in the screen.\r\n");
    printf(" play -- play wav file\r\n");
    printf(" ren -- rename a file\r\n");
}

/*******************************************************************************
* 函数名称：parseCommand()
* 函数功能：输入命令处理
* 入口参数：*cmd    命令字符串
*******************************************************************************/
void parseCommand(uint8 *cmd){
    char *c, *p1, *p2;
    c = strtok((char*)cmd, " ");
    p1 = strtok(NULL, " ");
    p2 = strtok(NULL, " ");
```

```
if(strcmp(c, "help") == 0 || strcmp(c, "h") == 0
                                    || strcmp(c, "/?") == 0 || strcmp(c, "?") == 0){
        printHelpMsg();
 }
 else if(strcmp(c, "fs") == 0 || strcmp(c, "FS") == 0){
      printFileSystemInfo();
 }
 else if(strcmp(c, "dir") == 0 || strcmp(c, "DIR") == 0
                                    || strcmp(c, "ls") == 0 || strcmp(c, "LS") == 0){
      FAT_ListRootDirectory(buffer);
 }
 else if(strcmp(c, "ren") == 0 || strcmp(c, "REN") == 0
                                    || strcmp(c, "rename") == 0 || strcmp(c, "RENAME") == 0){
      if(p1 && p2){
           FAT_RenameFile(p1, p2);
      }
      else{
           printf("Usage: rename oldfile newfile\r\n");
      }
 }
 else if(strcmp(c, "del") == 0 || strcmp(c, "DEL") == 0
                                                   || strcmp(c, "delete") == 0 || strcmp(c, "DELETE") == 0){
      if(p1){
           FAT_Delete(p1);
      }
      else{
           printf("Usage: del filename\r\n");
      }
 }
 else if(strcmp(c, "open") == 0){
      if(p1){
           signed char h;
           signed int s;
           h = FAT_OpenRead(p1);
           if(h >= 0){
              memset(buffer, 0x00, 512);
              s = FAT_Read(h, buffer, 512);
              printf("Read %d bytes @ sector %04X.\r\n", s, temp1);
              debugPrintHexTable(512, buffer);
              s = FAT_Read(h, buffer, 512);
              debugPrintHexTable(512, buffer);
              fat_close(h);
           }
           else if(h == -1){
              printf("Invalid file path\r\n");
```

```
                }
            }
        }
        else if(strcmp(c, "play") == 0){
            if(p1){
                signed char h;
                signed int s;
                h = FAT_OpenRead(p1);
                if(h >= 0){
                    memset(buffer, 0x00, 512);
                    s = FAT_Read(h, buffer, 512);
                    if(s > 0){
                        wav_read_header(buffer);
                            printf("--------WAV file information-------------\r\n");
                            printf("Sample Rate is %d Hz\r\n", pwav.samplerate);
                            printf("Sample Bits is %d bits\r\n",pwav.samplebits);
                            printf("Channels is %d\r\n",pwav.channels);
                            printf("Total time is %dms\r\n", pwav.totalms);
                            printf("Data Length is %d Bytes\r\n", pwav.datalen);
                            printf("Total Frames is %d\r\n", pwav.totalframes);
                            printf("Frame Size is %d\r\n", pwav.framesize);
                            printf("-------------------------------------\r\n");
                            fat_close(h);
                    }
                }
                else if(h == -1){
                    printf("Invalid file path\r\n");
                }
                else if(h == -2){
                    printf("File '%s' does not exist.\r\n", p1);
                }
            }
        }
        else{
            printf("unknown command, type 'help' for more information.\r\n");
        }
}

/******************************************************************************
* 函数名称：main()
* 函数功能：主程序
******************************************************************************/
int main (){
    uint8 tmp;
    uint8 cmd[CLI_BUFFER_SIZE];
```

```
    uint8 cmdCounter = 0;
    PINSEL0 = 0x00000000;                // 设置所有引脚连接
    PINSEL1 = 0x00000000;
    IO0DIR = 0xFFffffff;                 // 设置 P0.0−P0.31 控制口为输出
#ifdef LCD_CONNECTED
    init_lcd ();
    clrram();
    disp_dz(0x3c,0x3c);
#endif
    UART0_Init();
    printf(SEND_STRING);
    FAT_Initialize(buffer);
    memset(cmd, 0, CLI_BUFFER_SIZE);
    printHelpMsg();
    putchar('>');
#ifdef LCD_CONNECTED
    clrram();
#endif
    while(1){
        tmp = gchar();
        if(tmp == '\r' || tmp == '\n'){
            pchar(tmp);
            parseCommand(cmd);
            memset(cmd, 0, CLI_BUFFER_SIZE);
            cmdCounter = 0;
            printf("\r\n>");
        }
        else if(tmp == '\b'){
            if(cmdCounter > 0){
                cmd[--cmdCounter] = 0;
                pchar(tmp);
            }
        }
        else{
            if(cmdCounter < CLI_BUFFER_SIZE){
                cmd[cmdCounter++] = tmp;
                putchar(tmp);
            }
        }
    }
}
```

本例中还包括其他一些模块程序，限于篇幅，这里不再列出，读者可以从本书附带的光盘中找到这些模块程序。

5.5 网络芯片 ENC28J60 与 LPC2138 的接口及应用

5.5.1 ENC28J60 的功能模块与引脚分布

ENC28J60 是 Microchip Technology（美国微芯科技公司）推出的带 SPI 接口的独立以太网控制器。ENC28J60 符合 IEEE802.3 的全部规范，采用了一系列数据包过滤机制以对传输数据包进行限制。它还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。与主 CPU 的通信通过两个中断引脚和 SPI 实现，数据传输速率高达 10Mbit/s。两个专用引脚用于连接 LED，进行网络活动状态指示。ENC28J60 应用电路十分简单，要将 CPU 连接到速率为 10Mbit/s 的以太网，只需 ENC28J60、两个脉冲变压器和一些无源元件即可。

ENC28J60 由以下主要功能模块组成：

- SPI 接口，充当主 CPU 与 ENC28J60 之间通信通道。
- 控制寄存器，用于控制和监视 ENC28J60。
- 双端口 RAM 缓冲器，用于接收和发送数据包。
- 判优器，当 DMA、发送和接收模块发出请求时，对 RAM 缓冲器的访问进行控制。
- 总线接口，对通过 SPI 接收的数据和命令进行解析。
- 介质访问控制器 MAC 模块，实现符合 IEEE802.3 标准的 MAC 逻辑。
- 物理层 PHY 模块，对双绞线上的模拟数据进行编码和译码。

该器件还包括其他支持模块，诸如振荡器、片内稳压器、电平变换器（提供可以接受 5V 电压的 I/O 引脚）和系统控制逻辑。

ENC28J60 采用 28 脚封装，封装形式有 SPDIP、SOIC、SSOP 及 QFN，图 5-18 所示为其引脚排列，引脚功能如表 5-17 所列。

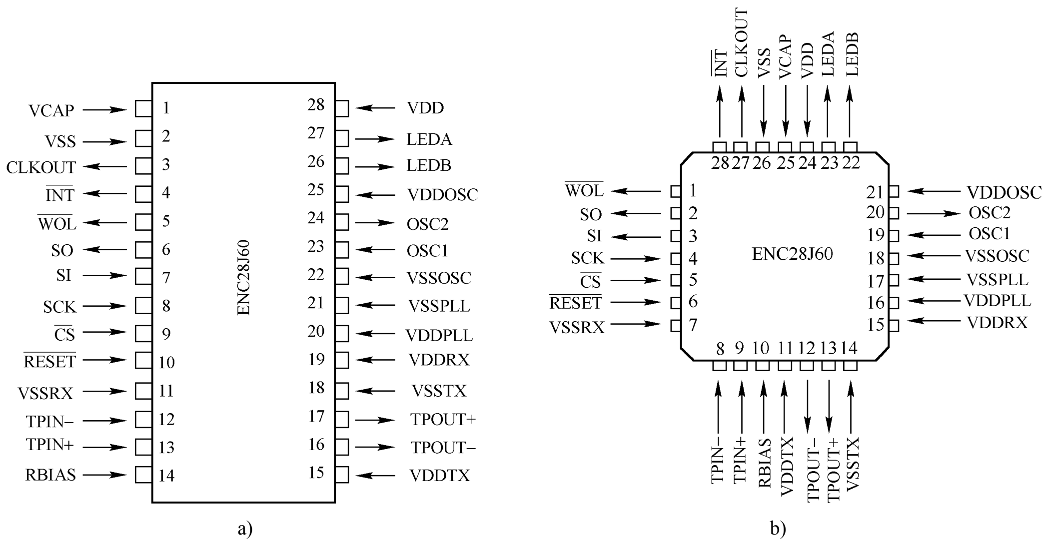


图 5-18 ENC28J60 引脚排列

a) 28 脚 SPDIP、SOIC、SSOP 封装 b) 28 脚 QFN 封装

表 5-17　ENC28J60 引脚说明

引脚名称	引脚编号		引脚类型	缓冲器类型	说　　明
	SPDIP、SOIC、SSOP	QFN			
VCAP	1	25	P	–	来自内部稳压器的 2.5V 输出。必须将此引脚通过一个 10μF 的电容连接到 VSSTX
VSS	2	26	P	–	参考接地端
CLKOUT	3	27	O	–	可编程时钟输出引脚①
$\overline{\text{INT}}$	4	28	O	–	INT 中断输出引脚②
$\overline{\text{WOL}}$	5	1	O	–	LAN 中断唤醒输出引脚②
SO	6	2	O	–	SPI 接口的数据输出引脚②
SI	7	3	I	ST	SPI 接口的数据输入引脚③
SCK	8	4	I	ST	SPI 接口的时钟输入引脚③
$\overline{\text{CS}}$	9	5	I	ST	SPI 接口的片选输入引脚③, ④
$\overline{\text{RESET}}$	10	6	I	ST	低电平有效器件复位输入③, ④
VSSRX	11	7	P	–	PHY RX 的参考接地端
TPIN−	12	8	I	ANA	差分信号输入
TPIN+	13	9	I	ANA	差分信号输入
RBIAS	14	10	I	ANA	PHY 的偏置电流引脚。必须将此引脚通过 2kΩ（1%）的电阻连接到 VSSRX
VDDTX	15	11	P	–	PHY TX 的正电源端
TPOUT−	16	12	O	–	差分信号输出
TPOUT+	17	13	O	–	差分信号输出
VSSTX	18	14	P	–	PHY TX 的参考接地端
VDDRX	19	15	P	–	PHY RX 的正 3.3V 电源端
VDDPLL	20	16	P	–	PHY PLL 的正 3.3V 电源端
VSSPLL	21	17	P	–	PHY PLL 的参考接地端
VSSOSC	22	18	P	–	振荡器的参考接地端
OSC1	23	19	I	DIG	振荡器输入
OSC2	24	20	O	–	振荡器输出
VDDOSC	25	21	P	–	振荡器的正 3.3V 电源端
LEDB	26	22	O	–	LEDB 驱动引脚⑤
LEDA	27	23	O	–	LEDA 驱动引脚⑤
VDD	28	24	P	–	正 3.3V 电源端

表中：I＝ 输入，O＝ 输出，P＝ 电源，DIG＝ 数字输入，ANA＝ 模拟信号输入，ST＝ 施密特触发器

① 引脚最大的电流驱动能力为 8mA。

② 引脚最大的电流驱动能力为 4mA。

③ 引脚可以接受 5V 的电压。
④ 引脚有一个连接到 VDD 的内部弱上拉电路。
⑤ 引脚最大的电流驱动能力为 12mA。

5.5.2 ENC28J60 主要特性

ENC28J60 主要特性如下：

1．以太网控制器特性

- IEEE802.3 兼容的以太网控制器。
- 集成 MAC 和 10BASE-T PHY。
- 接收器和冲突抑制电路。
- 支持一个带自动极性检测和校正的 10BASE-T 端口。
- 支持全双工和半双工模式。
- 可编程在发生冲突时自动重发。
- 可编程填充和 CRC 生成。
- 可编程自动拒绝错误数据包。
- 最高速度可达 10Mbit/s 的 SPI 接口。

2．缓冲器特性

- 8KB 发送/接收数据包双端口 SRAM。
- 可配置发送/接收缓冲器大小。
- 硬件管理的循环接收 FIFO。
- 字节宽度的随机访问和顺序访问（地址自动递增）。
- 用于快速数据传送的内部 DMA。
- 硬件支持的 IP 校验和计算。

3．介质访问控制器（MAC）特性

- 支持单播、组播和广播数据包。
- 可编程数据包过滤，并在以下事件的逻辑“与”和“或”结果为真时唤醒主机：

－ 单播目标地址。
－ 组播地址。
－ 广播地址。
－Magic Packet。
－ 由 64bit 哈希表定义的组目标地址。
－ 多达 64B 的可编程模式匹配（偏移量可由用户定义）。

- 环回模式。

4．物理层（MAC）特性

- 整形输出滤波器。
- 环回模式。

5．工作特性

- 两个用来表示连接、发送、接收、冲突和全/半双工状态的可编程 LED 输出。
- 使用两个中断引脚的七个中断源。

- 25MHz 时钟。
- 带可编程预分频器的时钟输出引脚。
- 工作电压范围：3.14～3.45V。
- TTL 电平输入。
- 温度范围：−40～+85°C（工业级），0～+70°C（商业级）（仅 SSOP 封装）。

5.5.3 ENC28J60 应用举例

Proteus 提供了 ENC28J60 以太网接口模型，它具备以下的仿真功能：

- SPI 接口。
- 8KB 的内部 RAM。
- 通用、MAC、MII 和 PHY 寄存器。
- 网络封包的发送和接收。
- DMA 操作（计算校验和，内存拷贝）。
- 包接收过滤（单播、多播、广播、Magic Packet、模式匹配、Hash 表、校验和）。
- 省电模式。
- LED 输出。
- 中断逻辑。
- 复位逻辑。

限制：

- 内置的自检控制器没有模型。

在使用该模型前，需要安装网络封包截取软件 WinPcap，可以从 WinPcap 的官方网络免费下载使用。

在使用 ENC28J60 发送和接收数据包前，必须先进行初始化设置。一些配置选项需要根据应用情况进行修改。通常应在复位后立即完成初始化任务，并且不再更改。在初始化过程中，需要配置一些 MAC 寄存器。这些寄存器只需配置一次，而编程的顺序并不重要。根据不同的应用，还需要对 3 个 PHY 模块寄存器中进行配置。

在接收数据包前，必须对接收缓冲器进行初始化。使用大量数据并经常交换数据包的应用，需要把大部分存储空间分配为接收缓冲器。需要保存已发数据或有一些数据包需要发送的应用，应分配较少的接收存储空间，所有未被用作接收缓冲器的存储空间都作为发送缓冲器。要发送的数据应写入未使用的空间。但在发送完一个数据包后，硬件会在存储器中数据包最后一个字节后写入一个 7B 的状态向量。因此，主控制器应在接收缓冲器的开始和每个包之间预留至少 7B，不需要对发送缓冲器进行特定的初始化。

下面的例子中移植了 Microchip 提供的一种 TCP/IP 协议栈，该协议栈服务于标准的、基于 TCP/IP 的应用程序（HTTP 服务器或邮件客户机等）。

Microchip TCP/IP 协议栈为多层结构，如图 5-19 所示。每层的实现代码唯一个独立的源程序文件，而服务和应用程序接口（API）则通过头文件或包含文件定义。与 TCP/IP 参考模型不同的是，Microchip TCP/IP 协议栈中各个层之间可以直接访问，而不需要逐层访问。关于一个层是否绕过相邻模块来获得所需的服务，主要根据服务开销的大小来决定。另外还添加了两个新模块：StackTask 和 ARPTask，StackTask 用于管理协议栈及其所有模块的操作，

而 ARPTask 用于管理地址解析协议（ARP）层的服务。

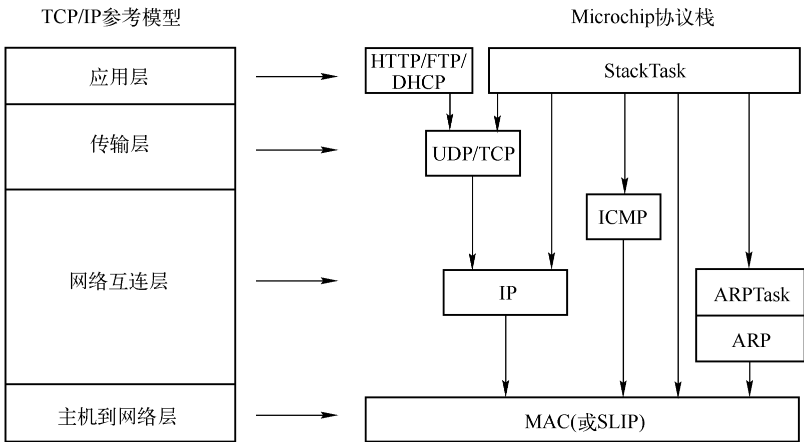


图 5-19　Microchip TCP/IP 协议栈的多层结构

TCP/IP 协议栈的某些层必须能异步执行某些定时操作。为了能满足这个要求，并与使用其服务的主应用程序保持相对独立，Microchip TCP/IP 协议栈使用了一种称为“协同式多任务处理”技术。在协同式多任务处理系统中，同时存在多个任务，每个任务执行完自己的作业后交回控制权，这样下个任务才能够执行作业。新增加的 StackTask 和 ARPTask 都是协同式任务。

Microchip TCP/IP 协议栈被设计为独立于任何操作系统，它可实现自己的协同式多任务处理，因此它可以独立使用而不必依赖其他操作系统。但是，使用 Microchip TCP/IP 协议栈的应用程序自身必须使用协同式多任务处理方法。需要注意的是，Microchip TCP/IP 协议栈没有提供一般 TCP/IP 协议栈中的全部模块，但是在需要时可以将它们作为单独的任务或模块来实现。

另外，本例中还采用了 Microchip 公司提供的 MPFS 文件系统，这是一个基于 ROM 的简单文件系统，适合于小型嵌入式环境。Microchip 公司提供了一个命令行应用工具：MPFS.exe，用于将网页转换为 MPFS 文件系统映像，该映像可以是 bin 格式，可以直接用来更新服务器的页面，也可以是 C 语言文件格式，需要与服务器源程序一起重新进行编译。

MPFS 命令格式为 MPFS.exe.\Webpages MPFSimg.c /c

如果希望更新网页内容，则需要进行如下操作：

1）把生成的文件"MPFSimg.c"放到 IAR 项目的 INC 目录下。

2）将#include "..\Include\Compiler.h" 改为 #include "Compiler.h"。

3）将 MPFS_Start[] 数组中的最后一组数据{0x04，0xffffff,...} 改为 {0x04，MPFS_END,...}。

4）重新编译整个项目。

例 5-7　图 5-20 所示为 LPC2138 与 ENC28J60 的接口应用电路。LPC2138 通过 SPI1（即 SSP）与 ENC28J60 接口，P0.16、P0.17、P0.18、P0.19、P0.29 和 P0.31 分别连接 ENC28J60 的 INT、SCK、SI、SO、RESET 和 CS 端。本例虚拟了一个 HTTP 服务器，它的 IP 地址是 192.168.0.128，可以通过任何一款网页浏览器进行访问。启动运行之前，先将计算

机的 IP 地址设置为 192.168.0.100，在图 5-19 所示电路中运行 Proteus 仿真程序，然后启动 IE 浏览器，在地址栏输入 192.168.0.128，即可访问虚拟 HTTP 服务器，如图 5-21 所示。

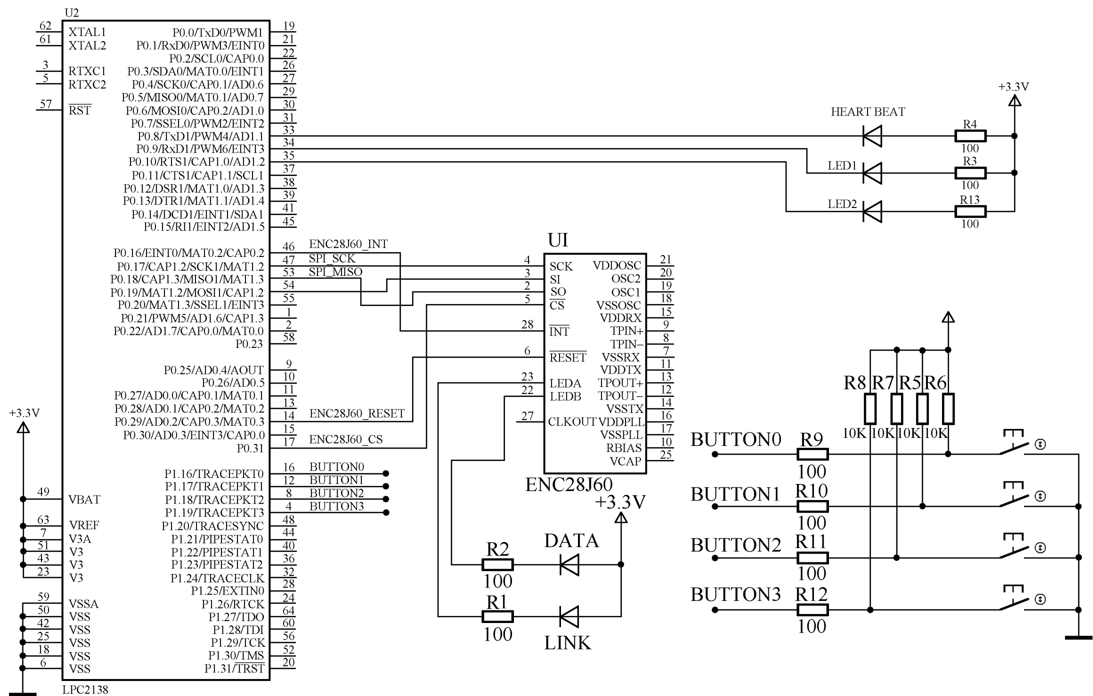


图 5-20　LPC2138 与 ENC28J60 的接口应用电路

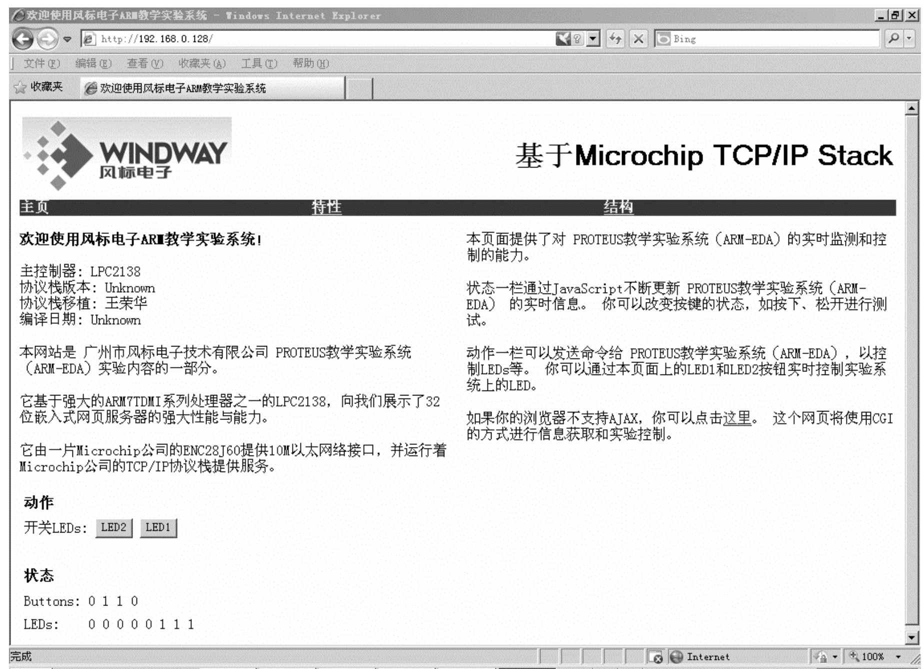


图 5-21　虚拟 HTTP 服务器网页

在 IE 浏览器中，可以通过超链接对 Proteus 仿真电路进行控制，单击网页上的 LED1 和 LED2 按钮，可以点亮或关闭仿真电路中的发光二极管，仿真电路中 4 个按键的状态也会反映到 IE 浏览器的网页上来。

本例最好使用两台电脑联网进行实验，在一台电脑上可能会遇到无法访问的问题。

本例中 ENC28J60 的驱动程序包括在 ENC28J60.C 模块中，具体如下：

```
#include "ENC28J60.h"
#include "config.h"
#include "StackTsk.h"
#include <string.h>

#define uint8_t    unsigned char
#define uint16_t unsigned short

#pragma pack(1)

static uint8_t Enc28j60Bank;
static uint16_t NextPacketPtr;

// MAC 层内部变量和标记.
WORD_VAL NextPacketLocation;
WORD_VAL CurrentPacketLocation;
BOOL WasDiscarded;

BUFFER CurrentTxBuffer;
BUFFER LastTXedBuffer;

#define ENC_SPI_CH        (0)

#define ETHER_IP      (0x00u)
#define ETHER_ARP   (0x06u)

typedef struct _ETHER_HEADER
{
    MAC_ADDR            DestMACAddr;
    MAC_ADDR            SourceMACAddr;
    WORD_VAL            Type;
} ETHER_HEADER;

extern uint8 SPI_SendData(uint8);

/**************************************************************************
*  函数名称：Delay_ms()
*  函数功能：毫秒延时程序
```

```
* 入口参数：延时时间
***************************************************************************/
void delay_ms(unsigned char ms)
{
      unsigned int t1, t2;
      for(t1=0;t1<ms;t1++)
     {
       for(t2=0;t2<100;t2++)
       {
             ;
       }
     }
}

/***************************************************************************
* 函数名称：Delay_us()
* 函数功能：微秒延时程序
* 入口参数：延时时间
***************************************************************************/
void _delay_us(unsigned char us)
{
      unsigned char len = 20;
for (;us > 0; us --)
      {
            for (;len > 0; len --);
                  len = 20;
      }
}

/***************************************************************************
* 函数名称：ENC28J60_Enable()
* 函数功能：使能 ENC28J60 的 SPI 接口，/CS = Low
***************************************************************************/
static void ENC28J60_Enable(void)
{
       IO0CLR |= ENC_CS;
}

/***************************************************************************
* 函数名称：ENC28J60_Disable()
* 函数功能：禁止 ENC28J60 的 SPI 接口，/CS = High。
***************************************************************************/
static void ENC28J60_Disable(void)
{
```

```
	IO0SET |= ENC_CS;
}

/*************************************************************************
* 函数名称：ENC28J60_Read_Reg()
* 函数功能：读寄存器
* 入口参数：opcode：操作码
    address：Reg 地址
* 出口参数：读取的数据
*************************************************************************/
uint8_t ENC28J60_ReadOp(uint8_t opcode, uint8_t address)
{
		uint8_t dat = 0;
		ENC28J60_Enable();
		dat = opcode | (address & ADDR_MASK);
		SPI_SendData(dat);
		dat = SPI_SendData(0);
		/*对于 MAC 和 MII（datasheet 29 页说明），需要执行一次空读操作。*/
		if(address & 0x80)
		{
		  dat = SPI_SendData(0);
		}
		// 释放片选信号 CS
		ENC28J60_Disable();
		return dat;
}

/*************************************************************************
* 函数名称：ENC28J60_Write_Reg()
* 函数功能：写寄存器
* 入口参数：延时时间
*************************************************************************/
void ENC28J60_WriteOp(uint8_t op, uint8_t address, uint8_t data)
{
		uint8_t dat = 0;
		ENC28J60_Enable();
		dat = op | (address & ADDR_MASK);			// 发布写入命令
		SPI_SendData(dat);
		dat = data;
		SPI_SendData(dat);					// 写入数据
		ENC28J60_Disable();
}

/*************************************************************************
```

```
* 函数名称：ENC28J60_ReadBuffer()
* 函数功能：读取缓冲区数据
* 入口参数：len 数据长度，* data 数据地址
*****************************************************************************/
void ENC28J60_ReadBuffer(uint16_t len, uint8_t* data)
{
        ENC28J60_Enable();
        SPI_SendData(ENC28J60_READ_BUF_MEM);            // 发布读取命令
        while(len)
        {
                len--;
                *data = (uint8_t)SPI_SendData(0);       //读取数据
                data++;
        }
        *data='\0';
        ENC28J60_Disable();
}

/*****************************************************************************
* 函数名称：ENC28J60_WriteBuffer()
* 函数功能：写入缓冲区数据
* 入口参数：len 数据长度，* data 数据地址
*****************************************************************************/
void ENC28J60_WriteBuffer(uint16_t len, uint8_t* data)
{
        ENC28J60_Enable();
        SPI_SendData(ENC28J60_WRITE_BUF_MEM);           // 发布写入命令
        while(len)
        {
                len--;
                SPI_SendData(*data);                    // 写入数据

                data++;
        }
        ENC28J60_Disable();
}

/*****************************************************************************
* 函数名称：ENC28J60_SetBank()
* 函数功能：设置数据组
* 入口参数：address 组地址
*****************************************************************************/
void ENC28J60_SetBank(uint8_t address)
{
```

```
		if((address & BANK_MASK) != Enc28j60Bank)		// 如果需要的话进行组设置
		{
			ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, (ECON1_BSEL1|
ECON1_BSEL0));
			ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, (address & BANK_
MASK)>>5);
			Enc28j60Bank = (address & BANK_MASK);
		}
	}

	/*****************************************************************************
	* 函数名称：ENC28J60_Read()
	* 函数功能：读取 ENC28J60
	* 入口参数：address 读取地址
	*****************************************************************************/
	uint8_t ENC28J60_Read(uint8_t address)
	{
		ENC28J60_SetBank(address);				// 调用设置组函数
		return ENC28J60_ReadOp(ENC28J60_READ_CTRL_REG, address);	// 读取数据
	}

	/*****************************************************************************
	* 函数名称：ENC28J60_Write()
	* 函数功能：写入 ENC28J60
	* 入口参数：address 写入地址，data 写入数据
	*****************************************************************************/
	void ENC28J60_Write(uint8_t address, uint8_t data)
	{
		ENC28J60_SetBank(address);				// 调用设置组函数
		// do the write
		ENC28J60_WriteOp(ENC28J60_WRITE_CTRL_REG, address, data);	// 写入数据
	}

	/*****************************************************************************
	* 函数名称：ENC28J60_PhyWrite()
	* 函数功能：写入 ENC28J60 PHY
	* 入口参数：address PHY 地址，data 写入数据
	*****************************************************************************/
	void ENC28J60_PhyWrite(uint8_t address, uint16_t data)
	{
		ENC28J60_Write(MIREGADR, address);			// 设置 PHY 寄存器地址
		ENC28J60_Write(MIWRL, data);				// 写入 PHY 数据
		ENC28J60_Write(MIWRH, data>>8);
		while(ENC28J60_Read(MISTAT) & MISTAT_BUSY){	// 等待写入结束
```

```
                    _delay_us(15);
            }
    }

    /*************************************************************************
    * 函数名称：ENC28J60_CLKOut()
    * 函数功能：写入 ENC28J60  时钟
    * 入口参数：clk  时钟数据
    *************************************************************************/
    void ENC28J60_CLKOut(uint8_t clk)
    {
          ENC28J60_Write(ECOCON, clk & 0x7);
    }

    /*************************************************************************
    * 函数名称：ENC28J60_Init()
    * 函数功能：ENC28J60 初始化
    * 入口参数：macaddr mac 地址
    *************************************************************************/
    void ENC28J60_Init(uint8_t* macaddr)
    {
        IO0CLR |=    ENC_RESET;                          //硬件复位
        Delayms(10);
        IO0SET|=ENC_RESET;
        Delayms(100);

       ENC28J60_WriteOp(ENC28J60_SOFT_RESET, 0, ENC28J60_SOFT_RESET); // ENC28J60 系统
复位

       /*  复位延时，至少等待 300μs 以使 PHY 稳定。要判断延时何时可以结束，主控制器可以
       查询 ESTAT.CLKRDY  位并等待它置 1 */
       delay_ms(250);
        ENC28J60_CLKOut(CLKDISABLE);               //  禁止 CLK 引脚输出信号

        /* LEDs  配置, LEDA=显示发送和接收活动, LEDB=显示链接状态和发送/接收活动  */
        ENC28J60_PhyWrite(PHLCON,0x7D6);

       /*  检查  CLKRDY  位，等待复位完成  */
       while(!(ENC28J60_Read(ESTAT) & ESTAT_CLKRDY));

       /*  对 Bank 0 寄存器进行初始化设置  */
       /*  初始化接收缓冲区  */
       /* 16 位的数据，必须先写低字节  */
       /*  设置接收缓冲区的开始地址  */
```

```
NextPacketPtr = RXSTART_INIT;
NextPacketLocation.Val = RXSTART_INIT;

 /* 接收缓冲区起始地址 Rx start */
 ENC28J60_Write(ERXSTL, RXSTART_INIT&0xFF);            // 低字节数据
 ENC28J60_Write(ERXSTH, RXSTART_INIT>>8);              // 高字节数据

 /* 接收缓冲区读指针 */
 ENC28J60_Write(ERXRDPTL, RXSTART_INIT&0xFF);
 ENC28J60_Write(ERXRDPTH, RXSTART_INIT>>8);

/* 接收缓冲区结束地址 RX end */
ENC28J60_Write(ERXNDL, RXSTOP_INIT&0xFF);
ENC28J60_Write(ERXNDH, RXSTOP_INIT>>8);

/* 发送缓冲区开始地址 TX start */
ENC28J60_Write(ETXSTL, TXSTART_INIT&0xFF);
ENC28J60_Write(ETXSTH, TXSTART_INIT>>8);

/* 发送缓冲区结束地址 TX end */
ENC28J60_Write(ETXNDL, TXSTOP_INIT&0xFF);
ENC28J60_Write(ETXNDH, TXSTOP_INIT>>8);

/* 对 Bank 1 寄存器进行初始化设置 */
/* 初始化包过滤器 */
/* 对于广播包，只接收 ARP 包 */
/* 对于其他的数据包，只接收发给本机（通过 MAC 地址确定）的数据包 */
/* ERXFCON：接收过滤器控制寄存器 */
/* UCEN：单播过滤器使能位 */
/* CRCEN：后过滤器 CRC 校验使能位 */
/* PMEN：格式匹配过滤器使能位 */
ENC28J60_Write(ERXFCON, ERXFCON_UCEN|ERXFCON_CRCEN|ERXFCON_PMEN);

/* 设置匹配模板 */
/* 类型ETH.DST */
/* ARP 广播 */
/* 0608 -- FF FF FF FF FF FF ->IP 检验和(Checksum)为 0xF7F9 */
ENC28J60_Write(EPMCSL, 0xf9);
ENC28J60_Write(EPMCSH, 0xf7);

/* 格式匹配屏蔽字 */
/* ARP 头
低字节--------------->高字节
目标地址 DA | 源地址 SA | 类型
```

```
  111111     |   000000   |   11
因此，ARP 头的屏蔽字为 0x303F
即 EPMM0=0x3F,EPMM1=0x30      */
ENC28J60_Write(EPMM0, 0x3f);
ENC28J60_Write(EPMM1, 0x30);

 /*  对 Bank 2 寄存器进行初始化  */
 /*  使能 MAC 接收
 MARXEN：MAC  接收使能位
 TXPAUS：暂停控制帧发送使能位，允许  MAC  发送暂停控制帧（用于全双工模式下的流量
控制）
 RXPAUS：暂停控制帧接收使能位，当接收到暂停控制帧时，禁止发送（正常操作）  */
ENC28J60_Write(MACON1, MACON1_MARXEN|MACON1_TXPAUS|MACON1_RXPAUS);

/* MAC 子系统复位解除  */
ENC28J60_Write(MACON2, 0x00);

/* MAC 控制寄存器 3
 PADCFG2:PACDFG0：自动填充和 CRC 配置位，001 =用 0 填充所有短帧至 60 个字节长，
 并追加一个有效的 CRC
 TXCRCEN：发送 CRC  使能位
 FRMLNEN：帧长度校验使能位
 FULDPX：MAC  全双工使能位      */
 ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, MACON3,
 MACON3_PADCFG0|MACON3_TXCRCEN|MACON3_FRMLNEN|MACON3_FULDPX);

/*  配置非背对背包间间隔寄存器  */
ENC28J60_Write(MAIPGL, 0x12);
ENC28J60_Write(MAIPGH, 0x0C);

/*  配置背对背包间间隔寄存器  */
ENC28J60_Write(MABBIPG, 0x12);

/*  设置允许接收或发送的最大帧长度，不能长于 MAX_FRAMELEN */
ENC28J60_Write(MAMXFLL, MAX_FRAMELEN&0xFF);
ENC28J60_Write(MAMXFLH, MAX_FRAMELEN>>8);

/*  对 Bank 3 寄存器进行初始化  */
 /*  将本地 MAC 地址写入 MAADR0:MAADR5 寄存器，注意：在这里是反序的  */
 ENC28J60_Write(MAADR5, macaddr[0]);
 ENC28J60_Write(MAADR4, macaddr[1]);
 ENC28J60_Write(MAADR3, macaddr[2]);
 ENC28J60_Write(MAADR2, macaddr[3]);
 ENC28J60_Write(MAADR1, macaddr[4]);
```

```
    ENC28J60_Write(MAADR0, macaddr[5]);

   /* 配置 PHY 工作模式
    PDPXMD：PHY  双工模式位
    1 = PHY 工作在全双工模式
    0 = PHY 工作在半双工模式
    PDPXMD 的默认复位值取决于 LEDB 引脚上外部连接的 LED 的极性
    要实现正确的双工操作，PHCON1.PDPXMD 位还必须与 MACON3.FULDPX 位的值匹配  */
    ENC28J60_PhyWrite(PHCON1, PHCON1_PDPXMD);

   /* 配置 PHY  控制寄存器 2
    HDLDIS：PHY  半双工环回禁止位      */
   ENC28J60_PhyWrite(PHCON2, PHCON2_HDLDIS);

   /* 选择 Bank 0*/
   ENC28J60_SetBank(ECON1);

   /* 使能中断 */
   ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, EIE, EIE_INTIE|EIE_PKTIE);

   /* 使能接收数据包 */
   ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_RXEN);
}

/****************************************************************************
* 函数名称：ENC28J60_GetRev()
* 函数功能：读取 ENC28J60 芯片版本
* 出口参数：ENC28J60 版本号
****************************************************************************/
uint8_t ENC28J60_GetRev(void)
{
    return(ENC28J60_Read(EREVID));
}

/****************************************************************************
* 函数名称：ENC28J60_PacketSend()
* 函数功能：发送 ENC28J60 数据包
* 入口参数：len 数据包长度，* packet 数据包地址指针
****************************************************************************/
void ENC28J60_PacketSend(uint16_t len, uint8_t* packet)
{
    ENC28J60_Write(EWRPTL, TXSTART_INIT&0xFF);          // 设置发送缓冲区指针
    ENC28J60_Write(EWRPTH, TXSTART_INIT>>8);
```

```
	ENC28J60_Write(ETXNDL, (TXSTART_INIT+len)&0xFF);    // 设置给定数据包的发送指针
	ENC28J60_Write(ETXNDH, (TXSTART_INIT+len)>>8);

	ENC28J60_WriteOp(ENC28J60_WRITE_BUF_MEM, 0, 0x00); // 写入每包控制字节

	ENC28J60_WriteBuffer(len, packet);                  // 将数据包复制到发送缓冲区

	ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_TXRTS);
	                                                    // 将数据发送到网络

	if( (ENC28J60_Read(EIR) & EIR_TXERIF) ){
		ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, ECON1_TXRTS);
	}
}

/*******************************************************************************
* 函数名称：ENC28J60_PacketReceive()
* 函数功能：接收 ENC28J60 数据包
* 入口参数：maxlen 数据包最大长度， * packet 数据包地址指针
* 出口参数：接收到的数据包长度
*******************************************************************************/
uint16_t ENC28J60_PacketReceive(uint16_t maxlen, uint8_t* packet)
{
	uint16_t rxstat;
	uint16_t len;
	if( ENC28J60_Read(EPKTCNT) ==0 ){
		return(0);
	}

	ENC28J60_Write(ERDPTL, (NextPacketPtr));            // 设置接收数据包的起始指针
	ENC28J60_Write(ERDPTH, (NextPacketPtr)>>8);

	NextPacketPtr  = ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0);
	                                                    // 读取下一个数据包指针
	NextPacketPtr |= ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0)<<8;

	len  = ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0);    // 读取数据包长度
	len |= ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0)<<8;
	len-=4;
	rxstat  = ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0);
	rxstat |= ENC28J60_ReadOp(ENC28J60_READ_BUF_MEM, 0)<<8;

	if (len>maxlen-1){                                  // 限制接收长度
		len=maxlen-1;
```

```
        }

        if ((rxstat & 0x80)==0){                          // 检查 CRC 及符号错误
                len=0;
        }
        else{
            ENC28J60_ReadBuffer(len, packet);             // 从接收缓冲区复制数据
        }

    ENC28J60_Write(ERXRDPTL, (NextPacketPtr));
    ENC28J60_Write(ERXRDPTH, (NextPacketPtr)>>8);
    ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON2, ECON2_PKTDEC);
    return(len);
}


/*******************************************************************************
 * 函数名称： ReadETHReg()
 * 函数功能： 读取 ETH 寄存器
 * 输入参数： Address ETH 控制寄存器地址
 * 出口参数： 读取的字节
*******************************************************************************/
static REG ReadETHReg(BYTE Address)
{
    REG r;
    ENC28J60_SetBank(Address);
    r.Val = ENC28J60_ReadOp(ENC28J60_READ_CTRL_REG, Address);
    return r;
}

/*******************************************************************************
 * 函数名称： MACPut()
 * 函数功能： 发布 MAC 数据
 * 输入参数： data    MAC 数据
*******************************************************************************/
void MACPut(uint8 data)
{
    ENC28J60_Enable();
    SPI_SendData(ENC28J60_WRITE_BUF_MEM);
    SPI_SendData(data);
    ENC28J60_Disable();
}

/*******************************************************************************
```

```
 * 函数名称： MACGet()
 * 函数功能： 获取 MAC 数据
 * 输出参数： 获取的 MAC 数据
*******************************************************************************/
uint8 MACGet()
{
    uint8 data;
    ENC28J60_Enable();
    SPI_SendData(ENC28J60_READ_BUF_MEM);
    data = SPI_SendData(0);
    ENC28J60_Disable();
    return data;
}

/*******************************************************************************
 * 函数名称： MACPutArray()
 * 函数功能： 发布 MAC 数组
 * 输入参数： *val 源字节指针，len MAC 数组长度
*******************************************************************************/
void MACPutArray(BYTE *val, WORD len)
{
    ENC28J60_Enable();
    SPI_SendData(ENC28J60_WRITE_BUF_MEM);
    while(len)
    {
        len--;
        SPI_SendData(*val);
        val++;
    }
    ENC28J60_Disable();
}

/*******************************************************************************
 * 函数名称： MACGetHeader()
 * 函数功能： 获取 MAC 头
 * 输入参数： * remote 储存源 MAC 地址的位置，*type 存储 MAC 内容的字节位置
 * 输出参数： 1 表示接收缓冲区有数据包等待接收
 *            0 表示接收缓冲区无数据包等待接收
*******************************************************************************/
BOOL MACGetHeader(MAC_ADDR *remote, BYTE* type)
{
     ENC_PREAMBLE header;

     /* 检查缓冲区有否存有网络封包 */
```

```
if( ENC28J60_Read(EPKTCNT) ==0 )
{
        return FALSE;
}

/* 保存当前封包的地址 */
CurrentPacketLocation.Val = NextPacketLocation.Val;

#ifdef PRINT_ENC_DEBUG_INFO
printf("NextPacketLocation = 0x%04X\r\n", NextPacketLocation.Val);
#endif

/*  把读指针指向当前封包的开始地址 */
ENC28J60_Write(ERDPTL, NextPacketLocation.v[0]);
ENC28J60_Write(ERDPTH, NextPacketLocation.v[1]);

/* 读 MAC 头部数据 */
ENC28J60_ReadBuffer(sizeof(header), (BYTE*)&header);

 #ifdef PRINT_ENC_DEBUG_INFO
 printf("Got a packet header:\r\n");
 debugPrintHexTable(sizeof(header), (BYTE*)&header);
 #endif

/* 检验 ENC28J60 返回的数据是否正确。但这不会经常发生，
有可能是 CPU 与 ENC28J60 SPI 通信错误产生的  */
if(header.NextPacketPointer > RXSTOP_INIT ||
                ((BYTE_VAL*)(&header.NextPacketPointer))->bits.b0 ||
                        header.StatusVector.bits.Zero ||
                        header.StatusVector.bits.CRCError ||
                        header.StatusVector.bits.ByteCount > 1518 ||
                        !header.StatusVector.bits.ReceiveOk)
{
        printf("Hardware error!!!\r\n");
        printf("Need to reset the machine.\r\n");
        MACInit();
}

/* 保存下一封包的地址，这一地址由 ENC28J60 自动指定 */
NextPacketLocation.Val = header.NextPacketPointer;

 #ifdef PRINT_ENC_DEBUG_INFO
 printf("NextPacketLocation = 0x%04X\r\n", NextPacketLocation.Val);
 #endif
```

```
        /*  返回以太网帧的源 MAC 地址给调用者，该参数非常有用，可省略 ARP 过程来响应请求  */
        memcpy((void*)remote->v, (void*)header.SourceMACAddr.v, sizeof(*remote));

        /*  返回一个简化的以太网类型参数给调用者  */
        *type = MAC_UNKNOWN;
        if( (header.Type.v[0] == 0x08u) &&
        ((header.Type.v[1] == ETHER_IP) || (header.Type.v[1] == ETHER_ARP)) )
        {
                *type = header.Type.v[1];
        }

        /*  打开可丢弃标记  */
        WasDiscarded = FALSE;
      return TRUE;
}

/******************************************************************************
 *  函数名称：  MACDiscardTx()
 *  函数功能：  抛弃发送 MAC
 *  输入参数：  buffer    缓冲区
******************************************************************************/
void MACDiscardTx(BUFFER buffer)
{
#if MAC_TX_BUFFER_COUNT > 1
     if(buffer < sizeof(TxBuffers)/sizeof(TxBuffers[0]))
     {
                TxBuffers[buffer].Flags.bFree = TRUE;
                CurrentTxBuffer = buffer;
     }
#endif
}

/******************************************************************************
 *  函数名称：  MACDiscardRx()
 *  函数功能：  抛弃接收 MAC
******************************************************************************/
void MACDiscardRx(void)
{
        WORD_VAL NewRXRDLocation;

        /*  当前的 packet 已经被取消? */
        if( WasDiscarded )
                return;
```

```
        /* 设置已取消标记 */
        WasDiscarded = TRUE;

        NewRXRDLocation.Val = NextPacketLocation.Val - 1;

#if RXSTART_INIT== 0
       if(NewRXRDLocation.Val > RXSTOP_INIT)
#else
        if(NewRXRDLocation.Val < RXSTART_INIT|| NewRXRDLocation.Val > RXSTOP_INIT)
#endif
     {
        NewRXRDLocation.Val = RXSTOP_INIT;
     }

     ENC28J60_Write(ERXRDPTL,NewRXRDLocation.v[0]);
     ENC28J60_Write(ERXRDPTH, NewRXRDLocation.v[1]);

     /* packet 计数器减 1，显示已经完成对该 packet 的处理   */
     ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON2, ECON2_PKTDEC);
}


/*****************************************************************************
 * 函数名称：  MACInit()
 * 函数功能：  MAC 初始化
*****************************************************************************/
void MACInit()
{
   ENC28J60_Init((unsigned char*)&AppConfig.MyMACAddr);     //  初始化  ENC28J60
}

/*****************************************************************************
 * 函数名称：  MACGetArray()
 * 函数功能：  获取 MAC 数组
 * 入口参数：  *val   存储指针，len 从缓冲区读取的字节数
 * 出口参数：  读取的字节数
*****************************************************************************/
WORD MACGetArray(BYTE *val, WORD len)
{
   WORD i = 0;
   ENC28J60_Enable();
   SPI_SendData(ENC28J60_READ_BUF_MEM);
   while(len)
```

```
    {
            len--;
            i++;
            // read data
            *val = (uint8)SPI_SendData(0);
            val++;
    }
    *val='\0';
    ENC28J60_Disable();
return i;
}

/****************************************************************************
 * 函数名称： MACPutHeader()
 * 函数功能： 发布 MAC 头
 * 入口参数：  *remote 目标 MAC 地址的指针， type 写入的头类型，dataLen 数据长度
****************************************************************************/
void    MACPutHeader(MAC_ADDR *remote, BYTE type, WORD dataLen)
{
    #if MAC_TX_BUFFER_COUNT > 1
            WriteReg(EWRPTL, TxBuffers[CurrentTxBuffer].StartAddress.v[0]);
            WriteReg(EWRPTH, TxBuffers[CurrentTxBuffer].StartAddress.v[1]);
            dataLen += (WORD)sizeof(ETHER_HEADER) + TxBuffers[CurrentTxBuffer].StartAddress.
Val;
            TxBuffers[CurrentTxBuffer].EndAddress.Val = dataLen;
    #else
            ENC28J60_Write(EWRPTL, TXSTART_INIT&0xFF);
            ENC28J60_Write(EWRPTH, TXSTART_INIT>>8);

        dataLen += (WORD)sizeof(ETHER_HEADER) + TXSTART_INIT;

        ENC28J60_Write(ETXNDL, ((WORD_VAL*)&dataLen)->v[0]);
        ENC28J60_Write(ETXNDH, ((WORD_VAL*)&dataLen)->v[1]);
    #endif

      MACPut(0x00);
      MACPutArray((BYTE*)remote, sizeof(*remote));
      MACPutArray((BYTE*)&AppConfig.MyMACAddr, sizeof(AppConfig.MyMACAddr));

      MACPut(0x08);
      MACPut((type == MAC_IP) ? ETHER_IP : ETHER_ARP);
}

/****************************************************************************
```

```
 * 函数名称： MACFlush()
 * 函数功能： MACFlush
******************************************************************************/
void MACFlush(void)
{
#if MAC_TX_BUFFER_COUNT > 1
	BankSel(ETXSTL);
	WriteReg(ETXSTL, TxBuffers[CurrentTxBuffer].StartAddress.v[0]);
	WriteReg(ETXSTH, TxBuffers[CurrentTxBuffer].StartAddress.v[1]);
	WriteReg(ETXNDL, TxBuffers[CurrentTxBuffer].EndAddress.v[0]);
	WriteReg(ETXNDH, TxBuffers[CurrentTxBuffer].EndAddress.v[1]);
	LastTXedBuffer = CurrentTxBuffer;
	TxBuffers[CurrentTxBuffer].Flags.bTransmitted = TRUE;
#endif

  if( (ENC28J60_Read(EIR) & EIR_TXERIF) )
     {
          ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, ECON1_TXRTS);
     }
    ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_TXRTS);

    #if 0
     if(ENCRevID == 0x05)
{
	while(!(ReadETHReg(EIR).Val & (EIR_TXERIF | EIR_TXIF)));
	if(ReadETHReg(EIR).EIRbits.TXERIF)
	{
		WORD_VAL ReadPtrSave;
		WORD_VAL TXEnd;
		TXSTATUS TXStatus;
		BYTE i;

		ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, ECON1_TXRTS);

		ReadPtrSave.v[0] = ReadETHReg(ERDPTL).Val;
		ReadPtrSave.v[1] = ReadETHReg(ERDPTH).Val;

		TXEnd.v[0] = ReadETHReg(ETXNDL).Val;
		TXEnd.v[1] = ReadETHReg(ETXNDH).Val;
		TXEnd.Val++;

		ENC28J60_Write(ERDPTL, TXEnd.v[0]);
		ENC28J60_Write(ERDPTH, TXEnd.v[1]);
		MACGetArray((BYTE*)&TXStatus, sizeof(TXStatus));
```

```
			for(i = 0; i < 16; i++)
			{
				if(ReadETHReg(EIR).EIRbits.TXERIF && TXStatus.bits.LateCollision)
				{
					ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_TXRST);
					ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, ECON1_TXRST);
					ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, EIR, EIR_TXERIF | 
EIR_TXIF);

					ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_TXRTS);
					while(!(ReadETHReg(EIR).Val & (EIR_TXERIF | EIR_TXIF)));

					ENC28J60_WriteOp(ENC28J60_BIT_FIELD_CLR, ECON1, ECON1_TXRTS);

					ENC28J60_Write(ERDPTL, TXEnd.v[0]);
					ENC28J60_Write(ERDPTH, TXEnd.v[1]);
					MACGetArray((BYTE*)&TXStatus, sizeof(TXStatus));
				}
				else
				{
					break;
				}
			}

			ENC28J60_Write(ERDPTL, ReadPtrSave.v[0]);
			ENC28J60_Write(ERDPTH, ReadPtrSave.v[1]);
		}
	}
	#endif
	}

	/******************************************************************************
	 * 函数名称：MACCalcRxChecksum()
	 * 函数功能：计算 MAC 校验和
	 * 入口参数：offset 校验和起始偏移量，len 校验和总字节数
	 * 出口参数：16 位校验和
	******************************************************************************/
	WORD MACCalcRxChecksum(WORD offset, WORD len)
	{
		WORD_VAL temp;
		temp.Val = CurrentPacketLocation.Val + sizeof(ENC_PREAMBLE) + offset;
		if ( temp.Val > RXSTOP_INIT)
		{
```

```
            temp.Val -= RXSIZE;
        }

        ENC28J60_Write(EDMASTL, temp.v[0]);
        ENC28J60_Write(EDMASTH, temp.v[1]);

        temp.Val += len-1;
        if ( temp.Val > RXSTOP_INIT)
        {
            temp.Val -= RXSIZE;
        }
        ENC28J60_Write(EDMANDL, temp.v[0]);
        ENC28J60_Write(EDMANDH, temp.v[1]);

        ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_DMAST | ECON1_
CSUMEN);
        while(ENC28J60_Read(ECON1) & 0x20);

        temp.v[1] = ENC28J60_Read(EDMACSL);
        temp.v[0] = ENC28J60_Read(EDMACSH);

        return temp.Val;
    }

    /******************************************************************************
     * 函数名称：  MACSetRxBuffer()
     * 函数功能：  设置 MAC 接收缓冲区
     * 入口参数：  offset 偏移量
    ******************************************************************************/
    void MACSetRxBuffer(WORD offset)
    {
        WORD_VAL ReadPT;
        ReadPT.Val = CurrentPacketLocation.Val + sizeof(ENC_PREAMBLE) + offset;
        if ( ReadPT.Val > RXSTOP_INIT)
        ReadPT.Val -= RXSIZE;
        ENC28J60_Write(ERDPTL, ReadPT.v[0]);
        ENC28J60_Write(ERDPTH, ReadPT.v[1]);
        ENC28J60_Write(EWRPTL, ReadPT.v[0]);
        ENC28J60_Write(EWRPTH, ReadPT.v[1]);
    }

    /******************************************************************************
     * 函数名称：  MACSetTxBuffer()
     * 函数功能：  设置 MAC 发送缓冲区
```

```
 * 入口参数： buffer 缓冲区， offset 偏移量
*****************************************************************************/
void MACSetTxBuffer(BUFFER buffer, WORD offset)
{
    CurrentTxBuffer = buffer;

#if MAC_TX_BUFFER_COUNT > 1
     offset += TxBuffers[buffer].StartAddress.Val + 1 + sizeof(ETHER_HEADER);
#else
     offset += TXSTART_INIT + 1 + sizeof(ETHER_HEADER);
#endif

     ENC28J60_Write(ERDPTL, ((WORD_VAL*)&offset)->v[0]);
     ENC28J60_Write(ERDPTH, ((WORD_VAL*)&offset)->v[1]);
     ENC28J60_Write(EWRPTL, ((WORD_VAL*)&offset)->v[0]);
     ENC28J60_Write(EWRPTH, ((WORD_VAL*)&offset)->v[1]);
}

/*****************************************************************************
 * 函数名称： MACGetTxBuffer()
 * 函数功能： 获取 MAC 发送缓冲区
 * 入口参数： HighPriority  优先级
 * 出口参数： BUFFER
*****************************************************************************/
BUFFER MACGetTxBuffer(BOOL HighPriority)
{
     #if MAC_TX_BUFFER_COUNT > 1
          BUFFER i;
          if(HighPriority)
     #endif
     {
          return !ReadETHReg(ECON1).ECON1bits.TXRTS ? 0 : INVALID_BUFFER;
     }

     #if MAC_TX_BUFFER_COUNT > 1
          for(i = 1; i < MAC_TX_BUFFER_COUNT; i++)
          {
               if(TxBuffers[i].Flags.bFree)
               {
                    TxBuffers[i].Flags.bFree = FALSE;
                    TxBuffers[i].Flags.bTransmitted = FALSE;
                    return i;
               }
          }
```

```
		return INVALID_BUFFER;
#endif
}

/******************************************************************************
 * 函数名称： MACIsTxReady()
 * 函数功能： MAC 发送就绪
 * 入口参数： HighPriority   优先级
 * 出口参数： 1 表示无发送， 0 上次发送还未结束
******************************************************************************/
BOOL MACIsTxReady(BOOL HighPriority)
{
	#if MAC_TX_BUFFER_COUNT > 1
		BUFFER i;
		if(HighPriority)
	#endif
	{
		return (BOOL)!ReadETHReg(ECON1).ECON1bits.TXRTS;
	}

	#if MAC_TX_BUFFER_COUNT > 1
		if(CurrentTxBuffer == LastTXedBuffer)
	{
		return !ReadETHReg(ECON1).ECON1bits.TXRTS;
	}

	for(i = 1; i < MAC_TX_BUFFER_COUNT; i++)
	{
		if(TxBuffers[i].Flags.bFree)
		{
			return TRUE;
		}
	}
	return FALSE;
	#endif
}

/******************************************************************************
 * 函数名称： MACGetFreeRxSize()
 * 函数功能： 获取 MAC 自由接收大小
 * 出口参数： 当前接收缓冲区的大致大小
******************************************************************************/
WORD MACGetFreeRxSize(void)
{
```

```
	WORD_VAL ReadPT, WritePT;
	do {
			ReadPT.v[0] = ReadETHReg((BYTE)EPKTCNT).Val;
			WritePT.v[0] = ReadETHReg(ERXWRPTL).Val;
			WritePT.v[1] = ReadETHReg(ERXWRPTH).Val;
	} while(ReadETHReg((BYTE)EPKTCNT).Val != ReadPT.v[0]);

	ReadPT.v[0] = ReadETHReg(ERXRDPTL).Val;
	ReadPT.v[1] = ReadETHReg(ERXRDPTH).Val;

	if ( WritePT.Val > ReadPT.Val )
	{
		return (RXSTOP_INIT - RXSTART_INIT) - (WritePT.Val - ReadPT.Val);
	}
	else if ( WritePT.Val == ReadPT.Val )
	{
		return RXSIZE – 1;
	}
	else
	{
		return ReadPT.Val – WritePT.Val – 1;
	}
}

/*****************************************************************************
 * 函数名称：  CalcIPBufferChecksum()
 * 函数功能：  计算 MAC 缓冲区校验和
 * 入口参数：  len  计算校验和的总字节数
 * 出口参数：  16 位校验和
*****************************************************************************/
WORD CalcIPBufferChecksum(WORD len)
{
	WORD_VAL temp;
	if(len == 0u)
	{
		return 0xFFFF;
	}
	else if(len == 1u)
	{
		return ~(((WORD)MACGet())<<8);
	}

	temp.v[0] = ReadETHReg(ERDPTL).Val;
	temp.v[1] = ReadETHReg(ERDPTH).Val;
```

```
        ENC28J60_Write(EDMASTL, temp.v[0]);
        ENC28J60_Write(EDMASTH, temp.v[1]);

    #if RXSTART_INIT== 0
            if(temp.Val <= RXSTOP_INIT)
    #else
            if(temp.Val >= RXSTART_INIT && temp.Val <= RXSTOP_INIT)
    #endif
    {
            temp.Val += len−1;
            if(temp.Val > RXSTOP_INIT)
            {
                temp.Val −= RXSIZE;
            }
    }
    else
    {
            temp.Val += len−1;
    }
    ENC28J60_Write(EDMANDL, temp.v[0]);
    ENC28J60_Write(EDMANDH, temp.v[1]);
    ENC28J60_WriteOp(ENC28J60_BIT_FIELD_SET, ECON1, ECON1_DMAST | ECON1_
CSUMEN);
    while(ReadETHReg(ECON1).ECON1bits.DMAST);

    temp.v[0] = ReadETHReg(EDMACSL).Val;
    temp.v[1] = ReadETHReg(EDMACSH).Val;
    return temp.Val;
}
```

进入主程序后首先进行引脚配置，规定 ENC28J60 的片选、复位等端口信号，完成 SPI1 初始化，以及 ENC28J60 芯片的初始化，并实现 HTTP 服务器功能。

主程序文件 main.c 列表如下：

```
#include "config.h"
#include <stdio.h>
#include <string.h>
#include "StackTsk.h"
#include "LPCUart.h"
#include "HTTP.h"
#include "TICK.h"
#include "Helpers.h"

/*************************************************************************
*  名称：SPI0_Init()
```

```
* 功能：SPI 接口初始化，设置为主机模式
*******************************************************************************/
void SPI0_Init(void)
{
S0SPCCR = 0x08;      // 设置 SPI 时钟分频，SPI 时钟＝PCLK/SPCCR，SPCCR 必须大于或等于 8
S0SPCR = 0x20;       // MSTR=1，主模式选择
}

/*******************************************************************************
* 名称：SPI0_SendData()
* 功能：向 SPI 总线发送数据或者读取数据。
* 入口参数：data   待发送的数据
* 出口参数：返回值为读取的数据
*******************************************************************************/
uint8 SPI0_SendData(uint8 data)
{
      S0SPDR = data;
      while( 0==(S0SPSR&0x80));
      return(S0SPDR);
}

/**********************************************************************************
* 函数名称 ：SSP_Init()
* 函数功能 ：将 SSP 控制器初始化 SPI 接口，设置为主机
**********************************************************************************/
void   SSP_Init(void)
{
    SSPCR0 = (0x01 << 8) |       // SCR   设置 SPI 时钟分频
             (0x00 << 7) |       // CPHA  时钟输出相位，仅 SPI 模式有效
             (0x00 << 6) |       // CPOL  时钟输出极性，仅 SPI 模式有效
             (0x00 << 4) |       // FRF   帧格式 00=SPI，01=SSI，10=Microwire，11=保留
             (0x07 << 0);        // DSS   数据长度，0000-0010=保留，0011=4 位，0111=8 位，
                                 1111=16 位

    SSPCR1 = (0x00 << 3) |            // SOD   从机输出禁能，1=禁止，0=允许
             (0x00 << 2) |            // MS    主从选择，0=主机，1=从机
             (0x01 << 1) |            // SSE   SSP 使能，1=允许 SSP 与其他设备通信
             (0x00 << 0);             // LBM   回写模式

    SSPCPSR = 0x06;                   //  PCLK 分频值
    SSPIMSC = 0x07;                   // 中断屏蔽寄存器
    SSPICR   = 0x03;                  // 中断清除寄存器
}
```

```
/*********************************************************************************
* 函数名称：SSP_SendData()
* 函数功能：SSP 接口向 SPI 总线发送数据
* 入口参数：data 待发送的数据
* 出口参数：返回值为读取的数据
**********************************************************************************/
uint8    SSP_SendData(uint8 data)
{
   SSPDR = data;
   while( (SSPSR & 0x10) );
   return(SSPDR);
}

/***************************************************************************
* 名称：SPI_SendData()
* 功能：通过 SPI0 或 SPI1 向 SPI 总线发送数据或者读取数据
* 入口参数：data    待发送的数据
* 出口参数：返回值为读取的数据
****************************************************************************/
uint8 SPI_SendData(uint8 data)
{
#ifdef ENC28J60_USING_SSP
        return SSP_SendData(data);
#else
        return SPI0_SendData(data);
#endif
}

/* 以下用于其他栈，必须在主应用中定义并赋值 */
APP_CONFIG AppConfig =
{
        {MY_DEFAULT_IP_ADDR_BYTE1, MY_DEFAULT_IP_ADDR_BYTE2,
              MY_DEFAULT_IP_ADDR_BYTE3, MY_DEFAULT_IP_ADDR_BYTE4},
        {MY_DEFAULT_MAC_BYTE1,     MY_DEFAULT_MAC_BYTE2,     MY_DEFAULT_MAC_
BYTE3,
              MY_DEFAULT_MAC_BYTE4, MY_DEFAULT_MAC_BYTE5, MY_DEFAULT_ MAC_
BYTE6},
        {MY_DEFAULT_MASK_BYTE1, MY_DEFAULT_MASK_BYTE2,
              MY_DEFAULT_MASK_BYTE3, MY_DEFAULT_MASK_BYTE4},
        {MY_DEFAULT_GATE_BYTE1, MY_DEFAULT_GATE_BYTE2,
              MY_DEFAULT_GATE_BYTE3, MY_DEFAULT_GATE_BYTE4},
        {MY_DEFAULT_DNS_BYTE1, MY_DEFAULT_DNS_BYTE2,
              MY_DEFAULT_DNS_BYTE3, MY_DEFAULT_DNS_BYTE4},
        {0x01},     // 标志, 使能 DHCP
```

```
};

#define VERSION           "v3.75"          // TCP/IP 栈版本号

extern void ENC28J60_Init(uint8* macaddr);
extern void Simple_Server(void);

static void ProcessIO(void)
{
  // 在这里添加用户程序
}

/*****************************************************************************
* 名称：main()
* 功能：主程序
*****************************************************************************/
int   main(void)
{
    static TICK t = 0;
    uint8 ledStatus = 0;

    TargetResetInit();

    PINSEL0 = 0x00000000;
    PINSEL1 = 0x00000000;

    PINSEL0 |= 0x00000005;                    // 设置 I/O 连接到 UART0

#ifdef ENC28J60_USING_SSP
    PINSEL1 |= 0x000002A8;                    // 设置 I/O 连接到 SSP
    SSP_Init();
#else
    PINSEL0 |= 0x1500;                        // 设置 I/O 连接到 SPI0
    SPI0_Init();
#endif
    PINSEL2 = 0x00000004;                     // 设置 I/O 连接到 UART0

    TickInit();
    UART0_Init();

    IO0DIR |= LED0 | LED1 | LED2 | ENC_CS|ENC_RESET|LEDCON;
    IO0SET = LED0 | LED1 | LED2;

    StackInit();
```

```
    HTTPInit();

    while(1)
    {
        if ( TickGetDiff(TickGet(), t) >= TICK_SECOND/10 )
        {
            t = TickGet();
            if(ledStatus)
            {
              ledStatus = 0;
              IO0SET = (1<<8);
            }
            else
            {
              ledStatus = 1;
              IO0CLR = (1<<8);
            }
        }
        StackTask();

#if defined(STACK_USE_HTTP_SERVER)
        /* 这是 TCP 协议上的一个简单的网页服务器，它将监听 TCP 80 端口（由 mywwwport
           定义），并对远程请求进行响应。目前只支持一个套接字 */
        HTTPServer();
#endif
        ProcessIO();
  }
}

/* CGI 命令代码 */
#define CGI_CMD_DIGOUT          (0)
#define CGI_CMD_LCDOUT          (1)
#define CGI_CMD_RECONFIG        (2)
#define CMD_LED1                (0x0)
#define CMD_LED2                (0x1)

/* CGI 变量代码 */
#define VAR_LED0                (0x00)
#define VAR_LED1                (0x01)
#define VAR_LED2                (0x10)
#define VAR_LED3                (0x11)
#define VAR_LED4                (0x12)
#define VAR_LED5                (0x13)
#define VAR_LED6                (0x14)
```

```
#define VAR_LED7                    (0x15)
#define VAR_ANAIN_AN0               (0x02)
#define VAR_ANAIN_AN1               (0x03)
#define VAR_DIGIN0                  (0x04)     // 浏览器上按键 0
#define VAR_DIGIN1                  (0x0D)     // 浏览器上按键 1
#define VAR_DIGIN2                  (0x0E)     // 浏览器上按键 2
#define VAR_DIGIN3                  (0x0F)     // 浏览器上按键 3
#define VAR_STACK_VERSION           (0x16)
#define VAR_STACK_DATE              (0x17)
#define VAR_STROUT_LCD              (0x05)
#define VAR_MAC_ADDRESS             (0x06)
#define VAR_SERIAL_NUMBER           (0x07)
#define VAR_IP_ADDRESS              (0x08)
#define VAR_SUBNET_MASK             (0x09)
#define VAR_GATEWAY_ADDRESS  (0x0A)
#define VAR_DHCP            (0x0B)     // 网页更新时使用该变量
#define VAR_DHCP_TRUE       (0x0B)     // 生成网页时使用该变量
#define VAR_DHCP_FALSE      (0x0C)     // 生成网页时使用该变量

/*********************************************************************************
* 函数名称：HTTPExecCmd()
* 函数功能：HTTP 命令处理
* 入口参数：** argv   参数列表， argc   参数计数
*********************************************************************************/
#if defined(STACK_USE_HTTP_SERVER)

ROM char COMMANDS_OK_PAGE[] = "INDEX.CGI";
ROM char CONFIG_UPDATE_PAGE[] = "CONFIG.CGI";
ROM char CMD_UNKNOWN_PAGE[] = "INDEX.CGI";

#define COMMANDS_OK_PAGE_LEN       (sizeof(COMMANDS_OK_PAGE))
#define CONFIG_UPDATE_PAGE_LEN     (sizeof(CONFIG_UPDATE_PAGE))
#define CMD_UNKNOWN_PAGE_LEN       (sizeof(CMD_UNKNOWN_PAGE))

void HTTPExecCmd(BYTE** argv, BYTE argc)
{
    BYTE command;
    BYTE var;
#ifdef ENABLE_REMOTE_CONFIG
    BYTE CurrentArg;
    WORD_VAL TmpWord;
#endif
    static BYTE led1=0;
    static BYTE led2=0;
```

```
        command = argv[0][0] - '0';

        switch(command)
        {
        case CGI_CMD_DIGOUT:     // ACTION=0
           var = argv[1][0] - '0';
              switch(var)
              {
              case CMD_LED1:           // NAME=0
                                       // Toggle LED
                   if(led1)
                   {
                        IO0SET = LED1;
                        led1 = 0;
                   }
                   else
                   {
                        IO0CLR = LED1;
                        led1 = 1;
                   }
                   break;

              case CMD_LED2:           // NAME=1
                                       // Toggle LED.
                   if(led2)
                   {
                        IO0SET = LED2;
                        led2 = 0;
                   }
                   else
                   {
                        IO0CLR = LED2;
                        led2 = 1;
                   }
                   break;
               }
               break;
#if defined(USE_LCD)
        case CGI_CMD_LCDOUT:     // ACTION=1
         if(argc > 2)                  // Text provided in argv[2]
         {
               if(strlen(argv[2]) < 32)
               {
                     memset(LCDText, ' ', 32);
```

```
                strcpy(LCDText, argv[2]);
            }
            else
            {
                memcpy(LCDText, (void*)argv[2], 32);
            }

            LCDUpdate();
        }
        else                        // No text provided
        {
            LCDErase();
        }
            break;
#endif
#if ENABLE_REMOTE_CONFIG
 case CGI_CMD_RECONFIG:             // ACTION=2
        CurrentArg = 1;
        while(argc > CurrentArg)
        {
            TmpWord.byte.MSB = argv[CurrentArg][0];
            TmpWord.byte.LSB = argv[CurrentArg++][1];
          var = hexatob(TmpWord);

          if(CurrentArg >= argc)
            break;

          switch(var)
          {
          case VAR_SERIAL_NUMBER:
            AppConfig.SerialNumber.Val = atoi(argv[CurrentArg]);
            AppConfig.MyMACAddr.v[4] = AppConfig.SerialNumber.byte.MSB;
            AppConfig.MyMACAddr.v[5] = AppConfig.SerialNumber.byte.LSB;
               break;

          case VAR_IP_ADDRESS:
          case VAR_SUBNET_MASK:
          case VAR_GATEWAY_ADDRESS:
            {
                DWORD TmpAddr;

                    if(!StringToIPAddress(argv[CurrentArg], (IP_ADDR*)&TmpAddr))
                      break;
```

```
                if(var == VAR_IP_ADDRESS)
                {
                      if(TmpAddr != *(DWORD*)&AppConfig.MyIPAddr)
                            DHCPBindCount++;

                  memcpy((void*)&AppConfig.MyIPAddr,
                              (void*)&TmpAddr, sizeof(AppConfig.MyIPAddr));
                      }
                else if(var == VAR_SUBNET_MASK)
                  memcpy((void*)&AppConfig.MyMask,
                              (void*)&TmpAddr, sizeof(AppConfig.MyMask));
                else if(var == VAR_SUBNET_MASK)
                  memcpy((void*)&AppConfig.MyGateway,
                              (void*)&TmpAddr, sizeof(AppConfig.MyGateway));
          }
         break;

    case VAR_DHCP:
      if(AppConfig.Flags.bIsDHCPEnabled)
      {
            if(!(argv[CurrentArg][0]−'0'))
            {
                  AppConfig.Flags.bIsDHCPEnabled = FALSE;
            }
          }
          else
      {
            if(argv[CurrentArg][0]−'0')
            {
                        AppConfig.MyIPAddr.Val = 0x00000000ul;
                  AppConfig.Flags.bIsDHCPEnabled = TRUE;
                      AppConfig.Flags.bInConfigMode = TRUE;
                  DHCPReset();
            }
          }
         break;
}

      CurrentArg++;
    }

      SaveAppConfig();
     break;
#endif
```

```
    default:
        break;
    }

}
#endif

/*********************************************************************************
* 函数名称：HTTPGetVar()
* 函数功能：HTTP 服务函数向主应用的回呼
* 入口参数：var 变量标识，ref 当前回呼参考，val 变量缓冲区
* 出口参数：主应用需要的参考变量
*********************************************************************************/
#if defined(STACK_USE_HTTP_SERVER)
WORD HTTPGetVar(BYTE var, WORD ref, BYTE* val)
{
     static BYTE VarString[20];
#if ENABLE_REMOTE_CONFIG
     static BYTE VarStringLen;
     BYTE *VarStringPtr;

     BYTE i;
     BYTE *DataSource;
#endif

    switch(var)
    {
    case VAR_LED0:
        *val = LED0_IO ? '1':'0';
        break;
    case VAR_LED1:
        *val = LED1_IO ? '1':'0';
        break;
    case VAR_LED2:
        *val = LED2_IO ? '1':'0';
        break;
    case VAR_LED3:
        *val = LED3_IO ? '1':'0';
        break;
    case VAR_LED4:
        *val = LED4_IO ? '1':'0';
        break;
    case VAR_LED5:
```

```
        *val = LED5_IO ? '1':'0';
        break;
    case VAR_LED6:
        *val = LED6_IO ? '1':'0';
        break;
    case VAR_LED7:
        *val = LED7_IO ? '1':'0';
        break;
    case VAR_DIGIN0:
        *val = BUTTON0_IO ? '1':'0';
        break;
    case VAR_DIGIN1:
        *val = BUTTON1_IO ? '1':'0';
        break;
    case VAR_DIGIN2:
        *val = BUTTON2_IO ? '1':'0';
        break;
    case VAR_DIGIN3:
        *val = BUTTON3_IO ? '1':'0';
        break;

     case VAR_STACK_VERSION:
        if(ref == HTTP_START_OF_VAR)
      {
                strcpy((char*)VarString, VERSION);
      }
        *val = VarString[(BYTE)ref];
        if(VarString[(BYTE)ref] == '\0')
            return HTTP_END_OF_VAR;
        else if(VarString[(BYTE)++ref] == '\0' )
            return HTTP_END_OF_VAR;
        return ref;
  case VAR_STACK_DATE:
        if(ref == HTTP_START_OF_VAR)
        {
                    strcpy((char*)VarString, __DATE__ " " __TIME__);
        }
        *val = VarString[(BYTE)ref];
        if(VarString[(BYTE)ref] == '\0')
                return HTTP_END_OF_VAR;
              else if(VarString[(BYTE)++ref] == '\0' )
                return HTTP_END_OF_VAR;
        return ref;
```

```
#if ENABLE_REMOTE_CONFIG
    case VAR_MAC_ADDRESS:
        if ( ref == HTTP_START_OF_VAR )
        {
            VarStringLen = 2*6+5;
            i = 0;
            VarStringPtr = VarString;
            while(1)
            {
              *VarStringPtr++ = btohexa_high(AppConfig.MyMACAddr.v[i]);
              *VarStringPtr++ = btohexa_low(AppConfig.MyMACAddr.v[i]);
              if(++i == 6)
                   break;
              *VarStringPtr++ = '-';
            }
        }

       *val = VarString[(BYTE)ref];
        if ( (BYTE)++ref == VarStringLen )
            return HTTP_END_OF_VAR;
        return ref;

    case VAR_SERIAL_NUMBER:
        if ( ref == HTTP_START_OF_VAR )
        {
              itoa(AppConfig.SerialNumber.Val, VarString);
              VarStringLen = strlen(VarString);
        }

        *val = VarString[(BYTE)ref];

        if ( (BYTE)++ref == VarStringLen )
            return HTTP_END_OF_VAR;
        return ref;

    case VAR_IP_ADDRESS:
    case VAR_SUBNET_MASK:
    case VAR_GATEWAY_ADDRESS:
        if ( ref == HTTP_START_OF_VAR )
        {
              if(var == VAR_IP_ADDRESS)
                   DataSource = (BYTE*)&AppConfig.MyIPAddr;
              else if(var == VAR_SUBNET_MASK)
                   DataSource = (BYTE*)&AppConfig.MyMask;
```

```
                else if(var == VAR_GATEWAY_ADDRESS)
                    DataSource = (BYTE*)&AppConfig.MyGateway;

                VarStringPtr = VarString;
                i = 0;
                while(1)
                {
                    itoa((WORD)*DataSource++, VarStringPtr);
                    VarStringPtr += strlen(VarStringPtr);
                    if(++i == 4)
                        break;
                    *VarStringPtr++ = '.';
                }
                VarStringLen = strlen(VarString);
            }

            *val = VarString[(BYTE)ref];

            if ( (BYTE)++ref == VarStringLen )
                return HTTP_END_OF_VAR;
            return ref;

        case VAR_DHCP_TRUE:
        case VAR_DHCP_FALSE:
            if ( ref == HTTP_START_OF_VAR )
            {
                if((var == VAR_DHCP_TRUE) ^ AppConfig.Flags.bIsDHCPEnabled)
                    return HTTP_END_OF_VAR;

                VarStringLen = 7;
                memcpypgm2ram(VarString, (rom void *)"checked", 7);
            }

            *val = VarString[(BYTE)ref];
            if ( (BYTE)++ref == VarStringLen )
                return HTTP_END_OF_VAR;
            return ref;
#endif
    }
    return HTTP_END_OF_VAR;
}
#endif
```

本例中还包括其他一些程序模块，限于篇幅，这里不再列出，读者可以从本书附带的光

盘中找到这些模块程序。

复习思考题

1．设计 LPC2138 与点阵字符型液晶显示模块的接口，绘制原理电路并编制相应程序。

2．设计 LPC2138 与 12864 图型液晶显示模块的接口，绘制原理电路并编制相应程序。

3．设计 T6963C 点阵图型液晶显示模块的接口，绘制原理电路并编制相应程序。

4．设计一个利用 DS18B20 与 LPC2138 实现的数字温度计，绘制原理电路并编制相应程序。

5．设计一个利用 DS1302 与 LPC2138 实现的万年历，绘制原理电路并编制相应程序。

6．设计一个 LPC2138 与 SD 卡的接口，绘制原理电路并编制相应程序。

7．设计一个利用网络芯片 ENC28J60 与 LPC2138 的接口，绘制原理电路并编制相应程序。

第 6 章　μC/OS-II 在 LPC2138 上的移植与应用

6.1　μC/OS-II 简介

μC/OS-II 是一款源代码开放的实时操作系统内核，由美国 Micrium 公司开发，它是专为嵌入式应用设计的，可用于各类 8 位、16 位和 32 位微处理器或 DSP。该内核已有 10 余年应用史，在诸多领域得到了广泛应用。μC/OS-II 具有如下特点：

- 开放源代码。可以到 Micrium 官方网站上下载 μC/OS-II 的源代码，对于个人研究是免费的，但对于商业用户需要交纳一定的费用。
- 可移植。μC/OS-II 源代码绝大部分使用移植性很强的 ANSI C 编写，只有与微处理器硬件相关的部分使用汇编语言编写。汇编语言编写的部分已经压缩到最低限度，使 μC/OS-II 便于移植到其他微处理器上。目前，μC/OS-II 已经被移植到多种不同架构的微处理器上。
- 可固化。只要具备合适的软硬件工具，就可将 μC/OS-II 嵌入到产品中成为产品的一部分。
- 可裁剪。μC/OS-II 使用条件编译实现可裁剪，用户程序可以只编译自己需要的功能，而不编译不需要的功能，以减少 μC/OS-II 对代码空间和数据空间的占用。
- 可抢占。μC/OS-II 是完全可剥夺型的实时内核，总是运行就绪条件下优先级最高的任务。
- 多任务。μC/OS-II 可以管理 64 个任务，建议用户保留 8 个任务给 μC/OS-II 本身，这样，实际留给用户的应用程序最多可有 56 个任务。
- 可确定性。绝大多数 μC/OS-II 函数调用和服务的执行时间具有确定性，也就是说，用户总是能知道 μC/OS-II 的函数调用与服务执行了多长时间。
- 单独任务栈。μC/OS-II 的每个任务都有自己单独的任务栈，使用 μC/OS-II 的栈空间检验函数，可确定每个任务到底需要多少栈空间。
- 完备的系统服务。μC/OS-II 提供很多系统服务，例如信号量、互斥信号量、时间标志、消息邮箱、消息队列、块大小固定的内存的申请与释放、时间管理函数等。
- 完善的中断管理。中断可使正在执行的任务暂时挂起，如果优先级更高的任务被中断唤醒，则高优先级的任务在中断嵌套全部退出后立即执行。中断嵌套的层数可达 255 层。
- 稳定性与可靠性。μC/OS-II 提供了更多功能，该操作系统的质量得到了官方认可，可以在任何的应用中使用。另外，μC/OS-II 还得到了美国联邦航空管理局对商用飞机符合 RTCA DO-178B 标准的认证。

6.2 μC/OS-II 在 LPC2138 上的移植

μC/OS-II 可以简单地看作一个实时多任务调度器，在此基础上完善并添加了与多任务操作系统有关的一些服务，如信号量、邮箱和消息队列等。图 6-1 所示为 μC/OS-II 的体系结构以及它与系统硬件之间的关系。

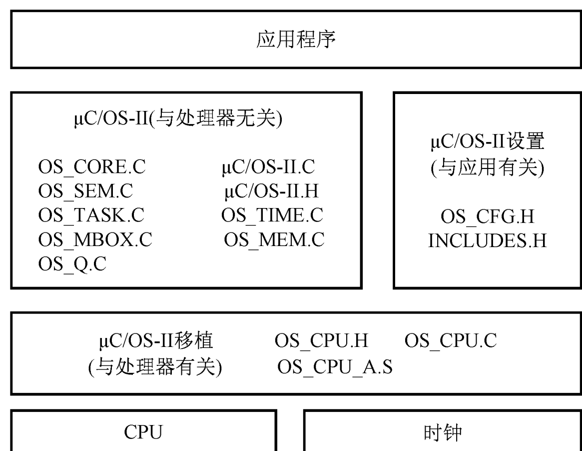


图 6-1　μC/OS-II 的硬件和软件体系结构

其中核心代码文件为 OS_CORE.C 操作系统内核功能、OS_FLAG.C 系统标志、OS_TIME.C 时钟、OS_TASK.C 多任务、OS_SEM.C 信号量、OS _MUTEX.C 互斥、OS_Q.C 消息队列、OS_MBOX.C 消息邮箱、OS_MEM.C 内存管理。

还有三个与处理器相关的文件：OS_CPU.H、OS_CPU_C.C、和 OS_CPU_ASM，其中的 OS_CPU_ASM 为汇编语言程序文件，它们也是将 μC/OS-II 移植到不同的 CPU 体系结构时需要修改的文件。

μC/OS-II 大部分代码都是用 C 语言编写的，因此具有良好的可移植性。移植工作绝大部分都集中在多任务切换的实现上，这部分代码主要用于保存和恢复与处理器相关寄存器的内容。另外具体移植时还要考虑采用哪一种编译器，针对 ARM 处理器核的 C 语言编译器有很多，这里选用的是 IAR EWARM 编译器。

移植一个操作系统到一个特定的 CPU 体系结构上并不是一件很容易的事情，它对移植者有以下要求：

- 对目标体系结构要有很深的了解。
- 对 OS 原理要有较深入的了解。
- 对所使用的编译器要有较深入的了解。
- 对需要移植的操作系统要有相当的了解。
- 对具体使用的芯片也要有一定的了解。

在移植 μC/OS-II 到 LPC2138 时，必须先把上述知识了解得比较透彻，这不单单是阅读资料就可以实现的，还需要不断地实践。其中，对操作系统的了解是最重要的，因为它的影响是全局性的，决定了移植代码的框架和功能。其他部分也很重要，它们在细节上影响代

码，并最终影响代码的正确性和可靠性。通常 µC/OS-II 中需要移植的部分如表 6-1 所列。

表 6-1 µC/OS-II 中需要移植的部分

移植内容	类型	所属文件	描述
BOOLEAN、INT8U、INT8S、INT16U、INT16S、INT32U、INT32S、FP32、FP64	数据类型	OS_CPU.H	与编译器无关的数据类型
OS_STK	数据类型	OS_CPU.H	堆栈的数据类型
OS_ENTER_CRITICAL 和 OS_EXIT_CRITICAL	宏	OS_CPU.H	开关中断的代码
OS_STK_GROWTH	常量	OS_CPU.H	定义堆栈的增长方向
OS_TASK_SW	函数	OS_CPU.H	任务切换时执行的代码
OSTaskStkInit()	函数	OS_CPU_C.C	任务堆栈初始化函数
OSInitHookBegin()、OSInitHookEnd()、OSTaskCreateHook()、OSTaskDelHook()、OSTaskSwHook()、OSTaskStatHook()、OSTCBInitHook()、OSTimeTickHook()、OSTaskIdleHook()	函数	OS_CPU_C.C	µC/OS-II 在执行某些操作时调用的用户函数，一般为空
OSStartHighRdy()	函数	OS_CPU_A. S	进入多任务环境时运行优先级最高的任务
OSIntCtxSw()	函数	OS_CPU_ASM	中断退出时的任务切换函数
OSTickISR	中断服务函数	OS_CPU_ASM	时钟节拍中断服务程序

实际上，还有一个 includes.h 文件需要关注，因为每一个应用都包含独特的 includes.h 文件。µC/OS-II 要求所有的 C 文件都要包含头文件 includes.h，这样使得用户项目中的每个 C 文件不用分别去考虑它实际上需要哪些头文件。使用 includes.h 的缺点是，它可能会包含一些实际不相关的头文件，这意味着每个文件的编译时间可能会增加，但却增强了代码的可移植性。用户可以按需要修改 includes.h 头文件，增加自己需要的内容，但必须加在文件末尾。

在本移植中另外增加了一个头文件 OS_CFG.H，要求所有用户程序必须包含 OS_CFG.H。在 OS_CFG.H 中包含 includes.h 和特定的头文件及配置项。而 µC/OS-II 的系统文件依然只是包含 includes.h，即 µC/OS-II 的系统文件完全不必改动。所有对配置的改变（包括头文件在内）均在 OS_CFG.H 中进行，而 includes.h 定下来后不必改动，这样，µC/OS-II 的系统文件需要编译的次数大大减少，编译时间也随之缩短。

6.2.1 编写 OS_CPU.H 文件

1．重新定义与编译器无关的数据类型

µC/OS-II 不使用 C 语言中的 short、int、long 等数据类型的定义，因为它们与处理器类型有关，隐含着不可移植性，代之以移植性强的整数数据类型，这样，既直观又可移植，不过这就成了必须移植的代码。

```
typedef unsigned char          BOOLEAN;                      /*  布尔变量     */
typedef unsigned char          INT8U;                        /*  无符号 8 位整型变量*/
typedef signed      char        INT8S;                        /*  有符号 8 位整型变量*/
typedef unsigned short  INT16U;                              /*  无符号 16 位整型变量*/
typedef signed      short       INT16S;                       /*  有符号 16 位整型变量*/
```

```
typedef unsigned int        INT32U;          /* 无符号 32 位整型变量*/
typedef signed    int       INT32S;          /* 有符号 32 位整型变量*/
typedef float               FP32;            /* 单精度浮点数（32 位长度）*/
typedef double              FP64;            /* 双精度浮点数（64 位长度）*/
typedef INT32U                  OS_STK;      /* 堆栈是 32 位宽度*/
```

2．使用软中断 SWI 作底层接口

移植中使用软中断指令 SWI 作为底层接口，使用不同的功能号区分不同的函数。软中断功能号分配如表 6-2 所列，未列出的为保留功能。用软中断作为操作系统的底层接口就需要在 C 语言中使用 SWI 指令。可以采用关键字__swi，用它声明一个不存在的函数，调用这个函数时，就在调用这个函数的地方插入一条 SWI 指令，并且可以指定功能号，同时，这个函数也可以有参数和返回值，其传递规则与一般函数一样。

表 6-2　软中断功能

功　能　号	接 口 函 数	说　　明
0x00	Void OS_TASK_SW(void)	任务级任务切换函数
0x01	_OSStartHighRdy(void)	运行优先级最高的任务，由 OSStartHighRdy
0x02	void OS_ENTER_CRITICAL(void)	关中断
0x03	Void OS_EXIT_CRITICAL(void)	开中断
0x80	Void ChangeToSYSMode(void)	任务切换到系统模式
0x81	Void ChangeToUSRMode(void)	任务切换到用户模式
0x82	Void TaskIsARM(INT8U prio)	任务代码是 ARM 代码
0x83	Void TaskIsTHUMB(INT8U prio)	任务代码是 Thumb 代码

3．设置堆栈的增长方向，堆栈由高地址向低地址增长

μC/OS-II 使用结构常量 OS_STK_GROWTH 中指定堆栈的生长方式：

置 OS_STK_GROWTH 为 0 表示堆栈从下往上长。

置 OS_STK_GROWTH 为 1 表示堆栈从上往下长。

虽然 ARM 处理器核对于两种方式均支持，但 ADS 的 C 语言编译器仅支持一种方式，即从上往下长，并且必须是满递减堆栈，所以 OS_STK_GROWTH 的值通常设置为 1：

```
#define OS_STK_GROWTH       1            /*   堆栈从上往下长 */
```

6.2.2　编写 OS_cpu_c.c 文件

1．OSTaskStkInit()函数

OS_CPU_C.C 文件是与处理器相关的 C 语言代码，一共包含 10 个函数：

```
OSTaskStkInit();              /* 初始化任务堆栈段 */
OSTaskCreateHook();           /* 任务建立钩子函数 */
OSTaskDelHook();              /* 任务删除钩子函数 */
OSTaskIdleHook();             /* 任务句柄钩子函数 */
OSTaskStatHook();             /* 任务状态钩子函数 */
OSTaskTickHook();             /* 任务节拍钩子函数 */
```

```
OSTimeTickHook();             /* 系统时钟节拍钩子函数 */
OSInitHookBegin();            /* 系统初始化钩子开始 */
OSInitHookEnd();              /* 系统初始化钩子结束 */
OSTCBInitHook();              /* 任务控制块初始化钩子 */
```

其中，只有 OSTaskStkInit()函数是必需的，其他 9 个必须声明，但不一定要有代码。另外还需要在 OS_CFG.H 文件中，将常量 OS_CPU_HOOKS_EN 置 1。

在编写 OSTaskStkInit()函数之前，必须先确定任务的堆栈结构，而任务的堆栈结构是与 CPU 的体系结构、编译器有密切的关联。

移植采用的堆栈结构如图 6-2 所示，据此写出函数 OSTaskStkInit()的代码：

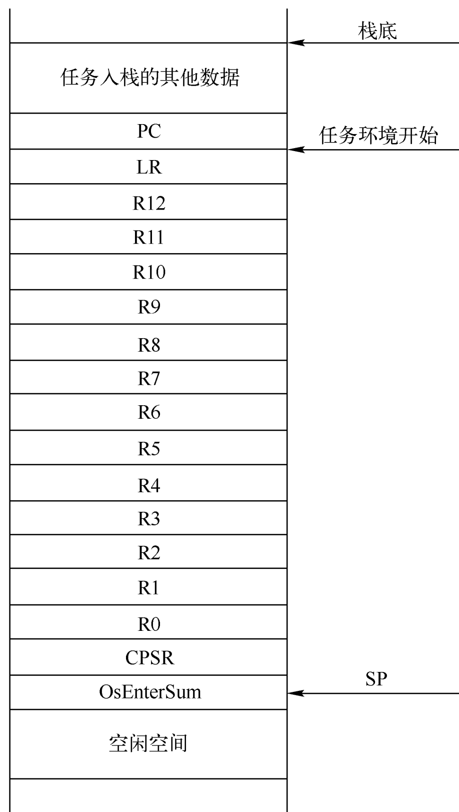


图 6-2　移植采用的堆栈结构

```
OS_STK *OSTaskStkInit (void (*task)(void *p_arg), void *p_arg, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;
    INT32U    task_addr;
    opt        = opt;                         /* opt 的作用是避免编译器警告 */
    stk        = ptos;                        /* 获取堆栈指针 */
    task_addr = (INT32U)task & ~1;            /* 在 Thumb 模式下屏蔽掉低位 */
    *(stk)       = (INT32U)task_addr;         /* 入口 */
    *(--stk)    = (INT32U)0x14141414L;        /* R14(LR) */
```

```
    *(--stk)  = (INT32U)0x12121212L;          /* R12 */
    *(--stk)  = (INT32U)0x11111111L;          /* R11 */
    *(--stk)  = (INT32U)0x10101010L;          /* R10 */
    *(--stk)  = (INT32U)0x09090909L;          /* R9 */
    *(--stk)  = (INT32U)0x08080808L;          /* R8 */
    *(--stk)  = (INT32U)0x07070707L;          /* R7 */
    *(--stk)  = (INT32U)0x06060606L;          /* R6 */
    *(--stk)  = (INT32U)0x05050505L;          /* R5 */
    *(--stk)  = (INT32U)0x04040404L;          /* R4 */
    *(--stk)  = (INT32U)0x03030303L;          /* R3 */
    *(--stk)  = (INT32U)0x02020202L;          /* R2 */
    *(--stk)  = (INT32U)0x01010101L;          /* R1 */
    *(--stk)  = (INT32U)p_arg;           /* R0  第一个参数使用 R0 传递。*/
    if ((INT32U)task & 0x01) {           /* 检查任务是运行在 Thumb 还是 ARM 模式 */
        *(--stk) = (INT32U)ARM_SVC_MODE_THUMB;    /* THUMB 模式 */
    } else {
        *(--stk) = (INT32U)ARM_SVC_MODE_ARM;      /* ARM 模式 */
    }
    return (stk);
}
```

2．软件中断异常 SWI 服务程序

软件中断异常 SWI 服务程序主要采用汇编语言程序 Os_cpu_a.asm 来实现，也可以采用 C 语言函数来实现，一般来说 C 语言处理函数代码结构比较简单，使用一个 switch 语句把各个功能分隔开，各个功能相对独立。

3．OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()函数

μC/OS-II 使用宏 OS_ENTER_CRITICAL()和 OS_EXIT_CRITICAL()来实现关中断和开中断。关中断和开中断是为了保护临界段代码。这些代码与处理器有关，是需要移植的代码。ARM 处理器关中断和开中断是通过改变程序状态寄存器 CPSR 中的相应控制位实现。由于使用了软件中断，程序状态寄存器 CPSR 保存到程序状态保存寄存器 SPSR 中，软件中断退出时会将 SPSR 恢复到 CPSR 中，所以程序只要改变程序状态保存寄存器 SPSR 中相应的控制位就可以了。改变这些位使用嵌入汇编实现，代码很简单。

4．OSStartHighRdy()函数

μC/OS-II 的启动多任务环境的函数叫做 OSStart()，用户在调用 OSStart()之前，必须已经建立了一个或更多任务。OSStart()最终调用函数 OSStartHighRdy()运行多任务启动前优先级最高的任务，OSStartHighRdy()的代码如下：

```
void OSStartHighRdy(void){
    _OSStartHighRdy();
}
```

这是调用软中断的 1 号功能，软中断的 1 号功能并没有在这里实现，它在 Os_cpu_asm 中实现。

5．移植增加的特定函数

根据 ARM 内核的特点和移植目标，增加了两个处理器模式转换函数：ChangeToSYSMode()、ChangeToUSRMode()和两个任务初始指令集设置函数 TaskIsARM()、TaskIsTHUMB()。它们都是通过软件中断指令 SWI 转换到系统模式，通过软件中断服务程序实现的。

处理器模式转换函数 ChangeToSYSMode()和 ChangeToUSRMode()使用软件中断功能 0x80 和 0x81 实现，其中函数 ChangeToSYSMode()把当前任务转换到系统模式，函数 ChangeToUSRMode()把当前任务转换到用户模式，它们可以在任何情况下使用。它们改变程序状态保留寄存器 SPSR 的相应位段，而程序状态保留寄存器会在软件中断退出时复制到程序状态寄存器 CPSR，任务的处理器模式就改变了。

任务可以使用 ARM 指令集或 Thumb 指令集运行，但是，任务建立时默认的只是一种指令集。如果任务使用的第一条指令与默认的指令集不同，将导致程序运行错误。为了纠正这个错误，移植中增加两个函数 TaskIsARM()和 TaskIsTHUMB()用于改变任务建立时默认的指令集。函数 TaskIsARM()用于声明指定优先级的任务的第一条指令是 ARM 指令，而函数 TaskIsTHUMB()则用于声明指定优先级的任务的第一条指令是 Thumb 指令，它们都有唯一的参数，即需要改变的任务的优先级。值得注意的是，这两个函数必须在相应的任务建立后但还没有运行时调用。这样，如果在低优先级的任务中创建高优先级的任务就十分危险了。解决这个问题的方法有三种：

1）高优先级任务使用默认的指令集。

2）改变函数 OSTaskCreateHook()使任务默认不是处于就绪状态，建立任务后调用函数 OSTaskResume()来使任务进入就绪状态。

3）建立任务时禁止任务切换，调用函数 TaskIsARM()或 TaskIsTHUMB()后再允许任务切换。

函数 TaskIsARM()和 TaskIsTHUMB()使用软件中断功能 0x82 和 0x83 实现。两个功能代码极其相似，代码也比较简单。首先，程序判断传递的参数（任务的优先级）是否在允许的范围内，然后获取任务的任务控制块(tcb)地址，接着判断指针是否有效，有效则改变指定任务的堆栈中存储的 CPSR 的 T 位。这两个函数仅在任务建立时使用。

6．Hook()函数

μC/OS-II 中有很多由用户编写的 Hook()函数，它们在本移植中全为空函数，用户可以根据 μC/OS-II 的要求进行修改。

6.2.3 编写 Os_cpu_asm 文件

1．软件中断的汇编接口

软中断的功能号包含在 SWI 指令中，程序通过读取该条指令的相应位段获得。由于 ARM 处理器具有两个指令集，其中指令的长度不同，SWI 指令的功能号的位段也不同，所以，程序先判断在进入软中断前处理器是在什么指令集状态，通过读取指令，取得指令中的功能号；然后，程序用功能号与 1 比较，当功能号小于 1 时就是 0 了，就跳转到任务切换函数 OS_TASK_SW()。当功能号等于 1 时，就跳转到第一次任务切换处，也就是_OSStartHighRdy。这两个功能不在 C 语言中实现，原因之一是因为它们需要明确的堆栈结构，这是 C 语

言不能提供的；原因之二是两个任务切换程序本身是使用汇编编写，且同在 os_cpu_asm 这个文件中，使用汇编跳转十分方便。

其他功能由软件中断的 C 语言处理函数处理，它有两个参数，第一个就是功能号，存放在 R0 中，第二个是保存参数和返回值的指针，也就是堆栈中存储用户函数 R0～R3 的位置，实质就是当前堆栈指针的值，它存放在 R1 中。

2．OS_TASK_SW()和 OSIntCtxSw()函数

OS_TASK_SW()函数是在 μC/OS-II 从低优先级任务切换到最高优先级任务时被调用的，它总是在任务级代码中被调用的。另一个 OSIntExit()函数被用来在 ISR 中使得更高优先级任务处于就绪状态时，执行任务切换功能，它最终调用 OSIntCtxSw()函数执行任务切换。OS_TASK_SW()是 SWI 软件中断的 0 号功能，通过调用 OSIntCtxSw()函数实现。此时的堆栈结构如图 6-3 所示。同时，R3 保存着 SPSR。这样，如果中断调用 OSIntCtxSw()时，需要相同的堆栈结构，R3 也要保存 SPSR，这需要中断服务程序保证。

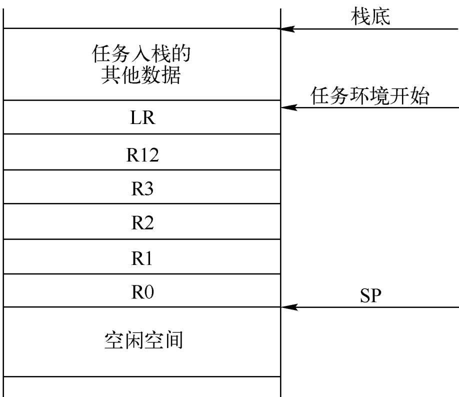


图 6-3　调用 OSIntCtxSw()时的栈结构图

OSIntCtxSw()的代码如下：

```
OSIntCtxSw
        LDR     R0, ??OS_TaskSwHook         ; 任务挂钩函数
        MOV     LR, PC
        BX      R0

        LDR     R4,??OS_PrioCur             ; 高优先级
        LDR     R5,??OS_PrioHighRdy
        LDRB    R6,[R5]
        STRB    R6,[R4]

        LDR     R4,??OS_TCBCur              ; 高级任务控制块
        LDR     R6,??OS_TCBHighRdy
        LDR     R6,[R6]
        STR     R6,[R4]

        LDR     SP,[R6]                     ; SP 指向任务控制块堆栈
```

```
                                        ; 恢复新任务上下文
        LDMFD    SP!, {R4}              ; 弹出新任务的 CPSR
        MSR      SPSR_cxsf, R4

        LDMFD    SP!, {R0-R12,LR,PC}^   ; 弹出新任务上下文
```

3．OSStartHighRdy()函数

μC/OS-II 启动多任务环境的函数叫作 OSStart()，用户在调用 OSStart()之前，必须已经建立了一个或更多任务。OSStart()通过调用函数 OSStartHighRdy()运行多任务启动前优先级最高的任务，它最终调用__OSStartHighRdy 来实现的。__OSStartHighRdy 的编写很简单，代码如下：

```
OSStartHighRdy
        LDR      R0, ??OS_TaskSwHook    ; 任务挂钩函数
        MOV      LR, PC
        BX       R0

        MSR      CPSR_cxsf, #0xD3       ; 切换到管理模式，并禁止 IRQ 及 FIQ 中断

        LDR      R4, ??OS_Running       ; 启动 OS 运行
        MOV      R5, #1
        STRB     R5, [R4]

                                        ; 切换到高优先级任务
        LDR      R4, ??OS_TCBHighRdy    ; 获得更高优先级任务的 TCB 地址
        LDR      R4, [R4]               ; 获得堆栈指针
        LDR      SP, [R4]               ; 切换到新任务

        LDR      R4,  [SP], #4          ; 弹出新任务的 CPSR
        MSR      SPSR_cxsf,R4
        LDMFD    SP!, {R0-R12,LR,PC}^   ; 弹出新任务上下文
```

6.3 编写 μC/OS-II 应用程序

移植的目的是为了在自己的系统使用 μC/OS-II，为此必须遵守 μC/OS-II 规范来编写自己的应用程序。与一般采用的“前后台方式”编程不同，μC/OS-II 采用“任务方式”来编程。

6.3.1 任务管理

μC/OS-II 的任务就是一个 C 语言函数，特别之处在于任务的结构是一种无限循环，并且任务不能被应用程序调用，因此，它的返回值必须为 void，为了便于应用程序向任务传递任何类型的数据，形式参数变量的类型是一个指向 void 的指针，它可以是一个变量、结构或函数的地址。任务示意代码如下：

```
void MyTask1(void *pdata) {
        ……                                    //任务的初始化
        *pdata = *pdata;                      //使用一次形式参量，以避免出现编译错误
                for ( ;   ; ) {               //超循环构成任务体
                可被中断的用户代码;
                OS_ENTER_CRITICAL();          //进入临界区（关中断）
                不可被中断的用户代码;
                OS_EXIT_CRITICAL();           //退出临界区（开中断）
                可被中断的用户代码;
        }
}
```

还有一种只执行一次就被删除的任务，它可以不是无限循环结构，示意代码如下：

```
void MyTask2(void *pdata){
        *pdata = *pdata;                  //使用一次形式参量，以避免出现编译错误
        用户代码;
        OSTaskDel(OS_PRIO_SELF);          //任务自我删除
}
```

需要注意的是，任务删除后，其代码依然驻留在 RAM 中，只是 μC/OS-II 将任务转入休眠状态，不再管理该段代码，而不是将代码真正地删除了，除非重新启动，否则该代码将不会再投入运行。

任务建立之后还必须要创建到 μC/OS-II 中，这通常是在 main()函数中，通过调用 μC/OS-II 提供的系统函数 OSTaskCreate()或 OSTaskCreateExt()实现的，示意代码如下：

```
#include "config.h"                                 //要求包含的头文件
OS_STK TaskStartStk[TASK_STK_SIZE];                 //分配任务的堆栈空间
OS_STK TaskStk[TASK_STK_SIZE];
void main( ) {
      … …
      OSInit( );                                    //μC/OS-II 初始化
      … …
      OSTaskCreate(MyTask1, (void *)0, &TaskStartStk[TASK_STK_SIZE－1], 4);
                                                    //将任务创建到 μC/OS-II 中
      OSTaskCreate(MyTask2, (void *)0, &TaskStartStk[TASK_STK_SIZE－1], 5);
      … …
      OSStart( );                                   //启动 μC/OS-II
}
```

编写 main()函数时需要注意，在使用 μC/OS-II 提供的任何功能之前，必须调用系统函数 OSInit()，它完成 μC/OS-II 的初始化并建立空闲任务。在开始多任务之前，必须通过调用系统函数 OSTaskCreate()或函数 OSTaskCreateExt()创建至少一个用户任务，最后通过调用系统函数 OSStart()，将控制权交给 μC/OS-II 内核，需要注意的是，在开始多任务之前，也就是调用 OSStart 之前，不允许打开时钟节拍中断，防止操作系统崩溃，时钟节拍中断的允许，是在 TargetInit 中完成的。

μC/OS-II 任务的优先级数是任务的唯一标识，系统函数 OSTaskCreate()或 OSTaskCreate Ext()的最后一个参数就是规定任务的优先级。μC/OS-II 可以管理多达 64 个任务，但是其中

两个最低优先级的已经分配给系统任务——空闲任务和统计任务了。优先级的值越小，任务的优先级就越高，不同的任务优先级不能相同，在准备就绪的任务中，总是优先级最高的任务得到 CPU 的使用权。实际上，应用程序最多能使用的优先级可以从 0 到 OS_LOWEST_PR1O-2，OS_LOWEST_PRIO（最低优先级）作为常数定义在 OS_CFG.H 文件中。

在任一特定时刻，任务一定处于休眠、就绪、挂起、被中断和运行五种状态之一。μC/OS-II 提供了若干系统函数，调用这些函数可以使任务从一种状态转变到另一种状态。

空闲任务 OSTaskIdle()是 μC/OS-II 中必不可少的系统函数，当没有其他任务进入就绪态时，该任务立即转入运行态。空闲任务的优先级永远设为最低，即 OS_LOWEST_PRIO 永远不被挂起，也不能被删除。

统计任务 OSTaskStat()也是 μC/OS-II 的系统函数之一，其功能是计算当前 CPU 的利用率，告诉用户应用程序使用了多少 CPU 时间。一旦将文件 OS_CFG.H 中的配置常数 OS_TASK_STAT_EN 置 1，这个任务就自动建立。它每秒钟运行一次，计算结果放在一个有符号的 8 位整数 OSCPUsage 中，表示格式是百分数，精确到 1%。

统计任务的使用方法是：用户在初始化时创建唯一一个任务，然后在这个唯一的任务中调用 OSStatInit()函数，同时再创建其他任务。使用统计任务时任务的创建方法与不使用统计任务时任务的创建方法不同，不使用统计任务时，任务既可以在初始化时创建，也可以在多任务启动后的任务中创建；使用统计任务时，初始化时只能创建唯一的一个任务，其他任务的创建只能在多任务启动之后创建。

任务的调度机制是 μC/OS-II 内核的核心，μC/OS-II 的调度器主要有两个功能：一是确定进入就绪态的任务中哪个优先级最高，二是进行任务切换。调度有两种方式：任务级的调度由系统函数 OSSched()完成，中断级的调度是由系统函数 OSIntExt()完成。μC/OS-II 任务调度所花的时间为常数，与应用程序中所建的任务数无关。μC/OS-II 一旦找到了准备就绪的最高优先级任务，就要进行任务切换。

μC/OS-II 提供 9 个任务管理函数，如表 6-3 所列，任务管理函数定义在 OS_TASK.C 文件中。

表 6-3　任务管理函数

任务管理函数	功　　能	参　　数
OSTaskChangePrio ()	改变任务的优先级	任务旧的优先级、任务新的优先级
OSTaskCreate ()	创建任务	任务代码指针、传递参数指针、任务堆栈栈顶指针、任务优先级
OSTaskCreateExt ()	创建扩展任务	任务代码指针、传递参数指针、任务堆栈栈顶指针、任务优先级、优先级标识（与优先级相同）、任务堆栈栈底指针、堆栈长度（检验用）、扩展 TCB 数据指针、任务设定选项
OSTaskDel ()	删除任务	任务优先级
OSTaskDelReq ()	请求一个任务删除其他任务或自身	任务优先级
OSTaskResume ()	唤醒一个用 OSTaskSuspend()函数挂起的任务	任务优先级
OSTaskStkChk ()	检查任务堆栈状态	任务优先级、检验堆栈数据结构
OSTaskSuspend ()	无条件挂起一个任务	任务优先级
OSTaskQuery ()	获取任务信息	任务指针、保存数据结构指针

6.3.2　任务管理应用编程举例

例 6-1　在应用程序中，创建两个任务 Task1 和 Task2，通过 LCD 显示其运行状态。Task1 只是简单显示运行状态，Task2 启动 A-D 转换，并将转换结果输出到 LCD。μC/OS-II 启动后，运行 Task1 和 Task2，任务切换几次后，将 Task1 挂起，只运行 Task2，等一会再恢复 Task1，最后删除 Task1；可以看到，任务挂起之后还可以恢复运行，而任务删除之后则再也不能恢复了，因此，最后只剩下 Task2 在运行。图 6-4 所示为运行以 LPC2138 位核心的 Proteus 仿真电路，本节关于 μC/OS-II 应用程序的例子都将在此电路上运行。

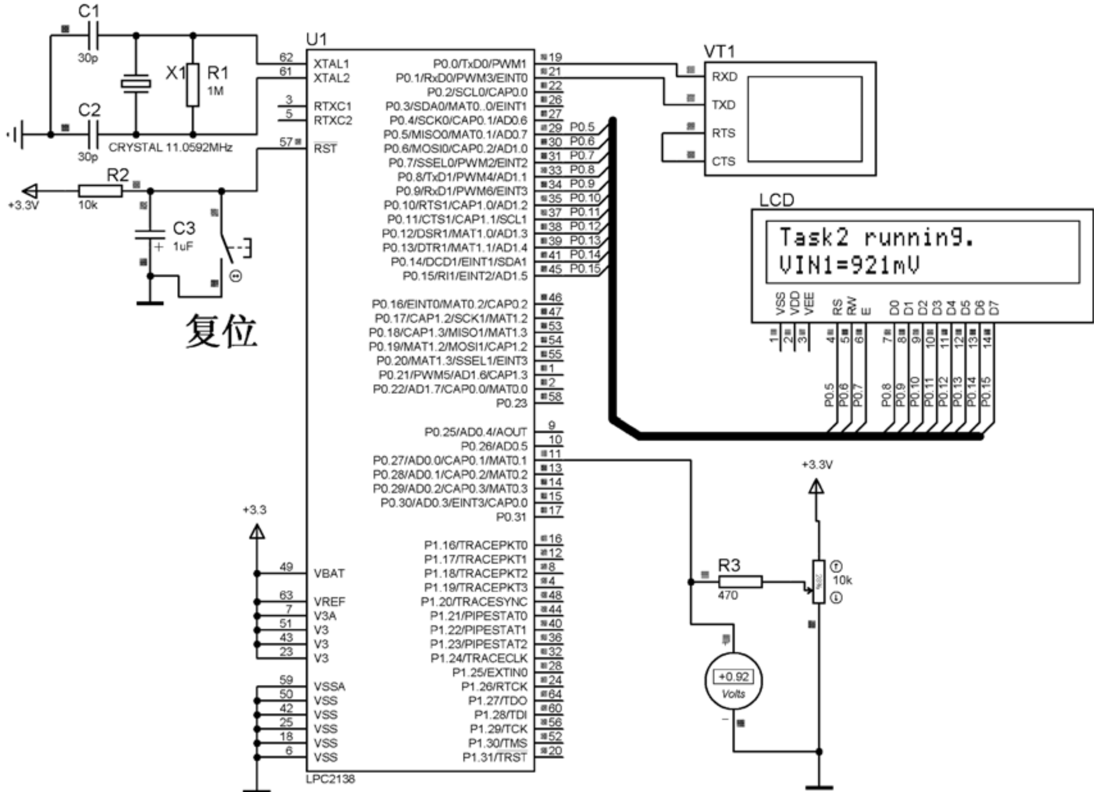


图 6-4　运行 μC/OS-II 应用程序的 Proteus 仿真电路

主程序文件 main.c 如下：

```
#include "config.h"
#include "target.h"                                    //这一句不能删除

#define     TaskStkLengh          200                  //定义任务堆栈
#define     TASK_STK_SIZE         200
#define     TASK_START_PRIO 1                          //定义起始任务优先级
#define     TASK_START_ID         0

OS_STK TaskStartStk[TaskStkLengh];
```

```
OS_STK Task1Stk[TASK_STK_SIZE];
OS_STK Task2Stk[TASK_STK_SIZE];
TASK_USER_DATA  TaskUserData[5];                    //扩展 TCB 数据类型

extern void Delay( uint32 ms );
extern void lcd_init(void);
extern void DisText(uint8 addr,uint8 *p);

static void TaskStart(void *pdata);                 //任务说明
void Task1(void *pdata);
void Task2(void *pdata);


/************************************************************************************
*  函数名称: main()
*  功能描述: 主函数，初始化 μC/OS-II 操作系统，创建任务，启动系统
************************************************************************************/
int   main(void){
      TargetResetInit();
  PINSEL1 = PINSEL1|0x01400000;            //  设置 P0.27、P0.28 连接到 AIN0、AIN1
  AD0CR = (1 << 0)                     |   // SEL = 1，选择通道 0
          ((Fpclk / 1000000 - 1) << 8) |   // CLKDIV = Fpclk / 1000000 – 1，
                                           //  即转换时钟为 1MHz
          (0 << 16)                    |   // BURST = 0，软件控制转换操作
          (0 << 17)                    |   // CLKS = 0，使用 11clock 转换
          (1 << 21)                    |   // PDN = 1，正常工作模式（非掉电转换模式）
          (0 << 22)                    |   // TEST1:0 = 00，正常工作模式（非测试模式）
          (1 << 24)                    |   // START = 1，直接启动 ADC 转换
          (0 << 27);                       // EDGE = 0（CAP/MAT 引脚下降沿触发 ADC 转换）

OSInit ();
OSTaskCreate(Task1, (void *)0, &Task1Stk[TASK_STK_SIZE - 1], 3);     //创建任务
OSTaskCreate(Task2, (void *)0, &Task2Stk[TASK_STK_SIZE - 1], 4);
OSTaskCreateExt(TaskStart,                                           //任务名
                (void *)0,                                           //参数指针
                &TaskStartStk[TaskStkLengh - 1],                     //堆栈栈顶
                TASK_START_PRIO,                                     //任务优先级
                TASK_START_ID,                                       //任务  ID
                &TaskStartStk[0],                                    //堆栈栈底
                TaskStkLengh,                                        //堆栈长度
                &TaskUserData[TASK_START_ID],                        //扩展 TCB 数据指针
                0);                                                  //禁止堆栈检测
OSStart ();                                                          //启动 μC/OS-II 多任务调度
return 0;
}
```

```
/********************************************************************************
*  函数名称: TaskStart()
*  功能描述: 起始任务，硬件目标板初始化
********************************************************************************/
void    TaskStart(void *pdata){
        pdata = pdata;                              //  避免编译警告
        UART0Init(57600);                           //  串行口初始化
        lcd_init();                                 //  液晶屏初始化
        for(;;){
                OSCtxSwCtr = 0;                     //清除上下文切换计数器
                OSTimeDly(70);                      //任务切换，交出 CPU 使用权
        }
}

/********************************************************************************
*  函数名称: Task2()
*  功能描述: 启动 A-D 转换并在 LCD 上显示结果，根据任务运行次数决定挂起、恢复和删除
                Task1
********************************************************************************/
void Task2(void *pdata){
   char       str[20];
   uint32    ADC_Data;
   INT8U j=0;
      for (;;)       {
              j++;
              DisText(0x80,"Task2 running.");
              DisText(0xc0,"                              ");
              AD0CR = (AD0CR&0x00FFFF00)|0x01|(1 << 24);
                                                        //  设置 A-D 通道 1，并进行第一次转换
              while( (AD0DR&0x80000000)==0 );           //  等待转换结束
              AD0CR = AD0CR | (1 << 24);                //  再次启运转换
              while( (AD0DR&0x80000000)==0 );           //  等待转换结束
              ADC_Data = AD0DR;                         //  读取 ADC 结果
              ADC_Data = (ADC_Data>>6) & 0x3FF;         //  提取 A-D 转换值
              ADC_Data = ADC_Data * 3300/1024;          //  数值转换
              sprintf(str, "VIN1=%dmV \r", ADC_Data);
              DisText(0xc0,str);
              printf("VIN1=%dmV \r\r\n",ADC_Data);
              if(j==2)              {
                     DisText(0xc0,"Task1 Suspend.");
                     DisText(0x80,"                              ");
                      printf("Task1 Suspend.\r\r\n");
                     OSTaskSuspend(3);                }
              if(j==5){
                     DisText(0xc0,"Task1 Resume.");
                     DisText(0x80,"                              ");
```

```
                printf("Task1 Resume.\r\r\n");
                OSTaskResume(3);            }
            if(j==6){
                OSTaskDel(3);
                DisText(0xc0,"Task1 Delete.");
                DisText(0x80,"                    ");
            }
            if(j==7)j=6;
            OSTimeDly(160);
    }
}

/*********************************************************************************
*  函数名称: Task1()
*  功能描述: 在 LCD 上显示任务运行状态
*********************************************************************************/
void Task1(void *pdata){
  while(1){
            DisText(0xc0,"Task1 running.");
            DisText(0x80,"                    ");
            printf("Task1 running.\r\r\n");
            OSTimeDly(120);
  }
}
```

6.4 任务的同步与通信

应用系统通常由多个任务组成，任务之间互相协作共同完成目标功能。例如任务对共享资源竞争，先采集数据才能处理数据等情况。各任务之间通过“任务通信”的方式，实现任务之间信息传递和同步控制，“任务通信”的载体就是事件。常用的事件有信号量、消息邮箱、消息队列、事件标志组等。对事件操作有创建、发送、请求和删除等。μC/OS-II 中以系统函数的方式向任务提供事件操作。限于篇幅，本节仅讨论信号量和消息邮箱。

6.4.1 信号量

信号量有两种类型：一种是只有 0 和 1 两种值的信号量，称为二值信号量，主要用于“独占式共享资源”的管理（例如：串行端口、打印机等）；另一种是可以有多种值的信号量，称为计数式信号量。计数式信号量的值的大小取决于信号量的数据类型，如若是 8 位整型变量，则其值可以是 0～255；若是 16 位整型变量，则其值可以是 0～65 535。

μC/OS-II 的信号量由两个部分组成：一个是信号量的计数值，范围是 0～65 535；另一个是由等待该信号量的任务组成的等待任务列表。当一个“事件”被占用时，其他请求该事件的任务暂时得不到事件的服务，处于等待状态。μC/OS-II 使用“等待任务表”来管理“事

件”；即：使用“等待任务表”对那些等待该事件的各个任务进行管理（记录等待该事件的任务并排序，任务等待事件的超时等）。每个事件都有一个“等待任务表”，用于完成事件对任务的驱动、超时等管理，其原理类似于任务就续表。任务等待事件的超时记录在 TCB 的 OSTCBDly 成员中，每个时钟节拍都会对其进行维护，当超时时间到时 μC/OS-II 强行将其转入就续状态。

信号量可以使用在如下场合：

- 允许一个任务与其他任务或中断同步。
- 取得共享资源的使用权。
- 标志事件的发生。

μC/OS-II 提供 6 个信号量管理函数，如表 6-4 所列，信号量管理函数定义在 OS_SEM.C 文件中。

表 6-4　信号量管理函数

信号量管理函数	功　　能	参　　数
OSSemAccept ()	无等待地请求一个信号量	--
OSSemCreate()	创建并初始化一个信号量	信号量初值
OSSemDel()	删除一个信号量	信号指针、删除条件、错误代码指针
OSSemPend ()	请求一个信号量	信号量指针、允许等待的时钟节拍、代码错误指针
OSSemPost ()	发送一个信号量	信号量指针
OSSemQuery ()	查询一个信号量的当前状态	信号量指针、状态数据结构指针

在使用信号量管理函数之前，必须将 OS_CFG.H 文件中相应的配置常量 OS_SEM_EN 设置为 1 或 0，以确定是编译还是裁剪该函数。原则上，中断服务子程序和任务可以共享信号量，但并不推荐这样使用，因为信号量一般用于任务级。如果非这样做不可，则中断服务子程序只能用来发送信号量。

信号量管理函数中，创建信号量、请求信号量和发送信号量最为重要。

1．创建信号量函数 OSSemCreate()

在使用信号量之前，必须先创建信号量。创建信号量的系统函数原型如下：

```
OS_EVENT     * OSSemCreate(
               INT16U cnt                //信号量计数初值
               )
```

OSSemCreate()函数用于建立一个信号量，并对信号量赋予计数初值。这个初值就是函数的参数 cnt，它可以是 0～65 535 中的任何值。信号量初值的设置规则如下：

- 如果信号量用来表示一个或多个事件的发生，那么该信号量的初值通常赋为 0。
- 如果信号量用于对共享资源的访问，那么该信号量的初值应赋为 1。
- 如果信号量用来表示允许访问 n 个相同的资源，那么该信号量的初值应赋为 n，并把信号量作为一个可计数的信号量使用。

OSSemCreate()函数返回的是分配给所建立的信号量的事件控制块的指针，如果没有可

用的事件控制块，则 OSSemCreate()函数返回空指针。

2．请求信号量函数 OSSemPend()

OSSemPend()函数请求信号量，函数原型如下：

```
void    OSSemPend(
                OS_EVENT    * pevent,        // 信号量指针指示申请目标
                INT16U    timeout,           // 等待限时，OS Tick  数
                INT8U       *err );          // 函数执行情况
```

OSSemPend()函数挂起当前任务，直到有其他的任务、中断置位信号量或者信号量超出等待的预期时间。如果在预期的时钟节拍内信号量被置位，那么 μC/OS-II 默认最高优先级的任务取得信号量并转入就绪。一个被 OSTaskSuspend()函数挂起的任务也可以接收信号量，但这个任务将一直保持挂起状态，直到通过调用 OSTaskResume()函数恢复任务的运行。

OSSemPend()函数有如下 3 个参数：

1）pevent：指向信号量的指针。指针的值可以在建立该信号量时得到。

2）timeout：指定的超时时钟节拍数。如果一个任务在经过了 timeout 参数指定的时钟节拍后还没有得到需要的信号量时，则转入就绪状态。如果 timeout 为 0，则表示任务将无限期地等待信号量。最大的超时等待时间为 65 535 个时钟节拍。

3）err：指向包含错误代码的变量指针。

OSSemPend()函数没有返回值。

3．发送信号量函数 OSSemPost()

OSSemPost()函数用于发送信号量，函数原型如下：

```
INT8U     OSSemPost(
                OS_EVENT    * pevent              //信号量指针指示释放目标
                )
```

OSSemPost()函数用于发送信号量。如果没有任务在等待信号量，那么 OSSem Post()函数使该信号量加 1 并返回；如果有任务在等待信号量，那么最高优先级的任务将得到信号量并进入就绪状态。任务调度函数将进行任务调度，决定当前运行的任务是否仍然为最高优先级的任务。从中断调用发送信号量的系统函数，不会发生任务切换，这是因为必须等到中断嵌套的最外层的 ISR 调用 OSIntExit()函数后，任务切换才能发生。

OSSemPost()函数只有 1 个参数 pevent，它是指向信号量的指针，指针的值可以在建立该信号量时得到。

OSSemPost()函数的返回值为下述内容之一：

- OS_NO_ERR：信号量成功置位，或者说成功发送。
- OS_SEM_OVF：信号量的值溢出。
- OS_ERR_EVENT_TYPE：pevent 不是指向信号量的指针。
- OS_ERR_PEVENT_NULL：pevent 是空指针。

通常请求信号量函数和发送信号量函数应该成对使用。

6.4.2 信号量应用编程举例

例 6-2 在应用程序中创建两个任务 Task1 和 Task2，它们都要使用 UART0 向外输出信息，由于 UART0 只有 1 个，不能同时被 Task1 和 Task2 使用，因此创建互斥信号量 semaphore 来进行约束，当 Task1 使用 UART0 时，Task2 就不能使用，反之亦然。μC/OS-II 启动后，通过虚拟终端可以清晰地看到在信号量 semaphore 约束下程序的运行状态。

主程序文件 main.c 如下：

```
#include "config.h"
#include "target.h"                              //这一句不能删除
#include <stdio.h>

#define      TaskStkLengh     200                //定义任务堆栈
#define      TASK_STK_SIZE       200
#define     TASK_START_PRIO        1             //定义起始任务优先级
#define     TASK_START_ID           0

OS_STK TaskStartStk[TaskStkLengh];
OS_STK Task1Stk[TASK_STK_SIZE];
OS_STK Task2Stk[TASK_STK_SIZE];
TASK_USER_DATA  TaskUserData[5];                 //扩展 TCB 数据类型

extern void Delay( uint32 ms );
extern void lcd_init(void);
extern void DisText(uint8 addr,uint8 *p);

static void TaskStart(void *pdata);  //任务说明
void Task1(void *pdata);
void Task2(void *pdata);

OS_EVENT *semaphore;                             //信号量说明

/*********************************************************************************
* 函数名称: main()
* 功能描述: 主函数，初始化 μC/OS-II 操作系统，创建任务和信号量，启动系统
*********************************************************************************/
int   main(void){
      TargetResetInit();
      PINSEL1 = PINSEL1|0x01400000;              // 设置 P0.27、P0.28 连接到 AIN0、AIN1
      AD0CR = (1 << 0)                  |        // SEL = 1，选择通道 0
              ((Fpclk / 1000000 - 1) << 8) |     // CLKDIV = Fpclk / 1000000 - 1，
// 即转换时钟为 1MHz
              (0 << 16)                 |        // BURST = 0，软件控制转换操作
              (0 << 17)                 |        // CLKS = 0，使用 11clock 转换
              (1 << 21)                 |        // PDN = 1，正常工作模式（非掉电转换模式）
              (0 << 22)                 |        // TEST1:0 = 00，正常工作模式（非测试模式）
```

```
            (1 << 24)                  |    // START = 1，直接启动 ADC 转换
            (0 << 27);                      // EDGE = 0（CAP/MAT 引脚下降沿触发 ADC 转换）
    OSInit ();
    semaphore=OSSemCreate(1);                                              //创建互斥信号量
    OSTaskCreate(Task1, (void *)0, &Task1Stk[TASK_STK_SIZE－1], 3);      //创建任务
    OSTaskCreate(Task2, (void *)0, &Task2Stk[TASK_STK_SIZE－1], 4);
    OSTaskCreateExt(TaskStart,                                   //任务名
                    (void *)0,                                   //参数指针
                    &TaskStartStk[TaskStkLengh－1],              //堆栈栈顶
                    TASK_START_PRIO,                             //任务优先级
                    TASK_START_ID,                               //任务 ID
                    &TaskStartStk[0],                            //堆栈栈底
                    TaskStkLengh,                                //堆栈长度
                    &TaskUserData[TASK_START_ID],                //扩展 TCB 数据指针
                    0);                                          //禁止堆栈检测
    OSStart ();                                                  //启动 μC/OS-II 多任务调度
    return 0;
}

/***********************************************************************************
*  函数名称: TaskStart()
*  功能描述: 起始任务,硬件目标板初始化
***********************************************************************************/
void   TaskStart(void *pdata){
    pdata = pdata;                              //  避免编译警告
    UART0Init(57600);                           //  串行口初始化
    lcd_init();
    for(;;){
        OSCtxSwCtr = 0;                         //清除上下文切换计数器
        OSTimeDly(60);                          //任务切换，交出 CPU 使用权
    }
}

/***********************************************************************************
*  函数名称: Task2()
*  功能描述: 启动 A-D 转换并在 LCD 上显示转换结果，在信号量约束下通过 UART0 输出信息
***********************************************************************************/
void Task2(void *pdata){
    char      str[20];
    uint32    ADC_Data;
    INT8U err;
    for (;;)      {
        OSSemPend(semaphore,0,&err);            //  请求信号量
        DisText(0x80,"Task2 running.");
        DisText(0xc0,"                ");
        AD0CR = (AD0CR&0x00FFFF00)|0x01|(1 << 24);
```

```
                                        // 设置 A-D 通道 1，并进行第一次转换
        while( (AD0DR&0x80000000)==0 );          // 等待转换结束
        AD0CR = AD0CR | (1 << 24);               // 再次启运转换
        while( (AD0DR&0x80000000)==0 );          // 等待转换结束
        ADC_Data = AD0DR;                        // 读取 ADC 结果
        ADC_Data = (ADC_Data>>6) & 0x3FF;        // 提取 A-D 转换值
        ADC_Data = ADC_Data * 3300/1024;         // 数值转换
        sprintf(str, "VIN1=%dmV \r", ADC_Data);
        UartSendString("Task2 get semaphore,running.\r\n");
        UartSendString(str);
        DisText(0xc0,str);
        OSSemPost(semaphore);                    // 释放信号量
        OSTimeDly(60);
    }
}

/*************************************************************************************
*  函数名称: Task1()
*  功能描述: 在信号量约束下通过 UART0 输出信息，同时在 LCD 上显示任务运行状态
*************************************************************************************/
void Task1(void *pdata){
    INT8U err;
    while(1){
        OSSemPend(semaphore,0,&err);                         // 请求信号量
        DisText(0xc0,"Task1 running.");
        DisText(0x80,"                    ");
        UartSendString("\rTask1 get semaphore,running.\r\n\r\n");
        OSSemPost(semaphore);                                // 释放信号量
        OSTimeDly(40);
    }
}
```

6.4.3 消息邮箱

消息邮箱的主要功能就是用于在任务间传递数据，它是一种通信机制，它能使任务或中断服务向另一个任务发送一个指针型的变量，这个指针指向一个包含指定“消息”的数据结构。消息邮箱发送的不是消息本身，而是消息的地址指针。消息邮箱具有如下特点：

- 消息邮箱中的内容是一个指向消息的指针，指针指向的内容即是消息。
- 消息邮箱为满时，消息邮箱只包含一个指向消息的指针；消息邮箱为空时，消息邮箱的指针指向 NULL。
- 消息邮箱只能接收和发送一则消息，消息邮箱为满时，将丢弃新消息，保留旧消息。

μC/OS-II 提供 7 个消息邮箱管理函数，如表 6-5 所列，信号量管理函数定义在 OS_MBOX.C 文件中。

表 6-5　消息邮箱管理函数

消息邮箱管理函数	功　　能	参　　数
OSMboxAccept ()	查询消息邮箱	消息邮箱指针
OSMboxCreate ()	创建并初始化一个消息邮箱	消息邮箱指针初值
OSMboxDel ()	删除一个消息邮箱	消息邮箱指针、删除条件、出错代码指针
OSMboxPend ()	请求一个消息邮箱	消息邮箱指针、允许等待的时钟节拍、代码错误指针
OSMboxPost ()	发送一个消息邮箱	消息邮箱指针、即将实际发送给任务的消息
OSMboxPostOpt ()	向邮箱发送一则消息	邮箱指针、消息、条件
OSMboxQuery ()	查询一个邮箱的当前状态	消息邮箱指针、状态数据结构指针

在使用信号量管理函数之前，必须将 OS_CFG.H 文件中相应的配置常量 OS_MBOX_EN 设置为 1 或 0，以确定是编译还是裁剪该函数。

用消息邮箱管理函数中，创建消息邮箱、请求消息邮箱和发送消息邮箱最为重要。

1．创建消息邮箱函数 OSMboxCreate ()

使用消息邮箱之前，必须先创建消息邮箱，创建消息邮箱的系统函数原型如下：

```
OS_EVENT    * OSMboxCreate(
                  void   * msg                    //  消息指针
            );
```

OSMboxCreate()函数只有一个参数 msg，指向消息的指针。该参数可以为 Null，也可以为非 Null。如果该指针非 Null，则建立的消息邮箱将含有消息。

当创建消息邮箱成功时，返回消息邮箱指针；消息邮箱创建失败时，返回 Null。

2．请求消息邮箱函数 OSMboxPend()

OSMboxPend()函数用于请求消息邮箱，也就是等待一个消息传送到消息邮箱，或取得一个消息数据，函数原型如下：

```
void   *OSMboxPend (
            OS_EVENT *pevent,                //消息邮箱指针
            INT16U timeout,                  //等待时限
            INT8U *err                       //函数执行信息
            );
```

OSMboxPend()函数具有如下特点：

① 如果调用时消息邮箱中已有消息，那么该消息被返回给调用者，并从消息邮箱中清除该消息。

② 调用时，如果消息邮箱中没有消息，则挂起当前任务直到得到需要的消息或等待超时期满。

③ 如果同时有多个任务等待同一个消息，μC/OS-II 将把消息交给优先级最高的任务并且恢复它的运行。

④ 一个由 OSTaskSuspend()函数挂起的任务也可以接收消息，但这个任务将一直保持挂起状态，直到通过调用 OSTaskResume()函数来恢复任务的运行。

⑤ OSMboxPend()函数只能在任务中调用，中断不能调用。

OSMboxPend()函数有如下 3 个参数：

1）pevent：指向接受消息的指针。指针的值在建立邮箱时得到。

2）timeout：超时定义。任务若在指定数目的时钟节拍后还没有得到消息，即恢复运行。如果该参数为零，则表示任务将持续地等待消息，直到收到消息才转入就绪。最大等待时间为 65 535 个时钟节拍。

3）err：指向包含错误码的变量指针。

OSMboxPend()函数的返回值如下：

当返回值 != Null 时，返回值就是一个预期消息的指针。

当返回值 =Null 时，意味着未得到消息，此时 μC/OS-II 执行 OS_Sched()；可能消息未准备好，或指示的事件出错、超时等，此时函数直接返回，用户可以查阅 *err 获知具体返回值状态。

3．发送消息邮箱函数 OSMboxPost()

系统函数 OSMboxPost( )用于向消息邮箱发送消息，函数原型如下：

```
INT8U       OSMboxPost(
                OS_EVENT    * pevent,                  // 消息邮箱指针
                 void    * msg                         // 消息指针
              );
```

OSMboxPost( )函数有如下两个参数：

1）Pevent：指向接收消息的消息邮箱指针。指针的值可以在建立该消息邮箱时得到；

2）msg：即将发送给任务的消息。这是一个要在任务之间传递的消息变量指针，不允许传递空指针，因为这意味着消息邮箱为空。

OSMboxPost()函数通过消息邮箱向任务发送消息，消息是指针型变量，消息的数据类型可能会因具体应用的不同而有所差异。该函数具有如下特点：

① 如果消息邮箱中已有消息，返回错误代码说明消息邮箱已满，函数立即返回调用者，并丢弃新消息。

② 如果消息邮箱中无消息，则有两种可能：

- 如果有任务在等待消息，最高优先级的任务将得到这个消息。如果等待消息的任务的优先级比函数的调用者优先级高，则这个高优先级任务得以恢复运行，调用者被挂起，发生一次任务切换。若从中断调用，则不发生任务切换。
- 如果没有任务在等待消息，则消息的指针被保存在消息邮箱中。

③ OSMboxPost()函数可以通过任务或中断调用。

OSMboxPost()函数的返回值 err 的错误代码可能是如下几种之一：

- OS_NO_ERR：消息成功地放到消息邮箱中。
- OS_MBOX_FULL：消息邮箱满，已经包含了其他消息。
- OS_ERR_EVENT_TYPE：pevent 不是指向消息邮箱的指针。
- OS_ERR_PEVENT_NULL：pevent 是空指针。
- OS_ERR_POST_NULL_PTR：用户试图发出空指针，根据规则，空指针无效。

6.4.4 消息邮箱应用编程举例

例 6-3 在应用程序中创建两个任务 Task1、Task2 以及 1 个消息邮箱。在 Task2 中启动 A-D 转换，并通过消息邮箱将 A-D 转换数据发送给 Task1，Task1 从消息邮箱中获取 A-D 转换数据，通过 UART0 向外输出。μC/OS-II 启动后，通过虚拟终端可以清晰地看到在消息邮箱的作用。

主程序文件 main.c 如下：

```
#include "config.h"
#include "target.h"
#include <stdio.h>

#define     TaskStkLengh    200                        // 定义任务堆栈
#define     TASK_STK_SIZE      200
#define     TASK_START_PRIO       1                    // 定义起始任务优先级
#define     TASK_START_ID         0

OS_STK TaskStartStk[TaskStkLengh];
OS_STK Task1Stk[TASK_STK_SIZE];
OS_STK Task2Stk[TASK_STK_SIZE];
TASK_USER_DATA  TaskUserData[5];                       // 扩展 TCB 数据类型

extern void Delay( uint32 ms );
extern void lcd_init(void);
extern void DisText(uint8 addr,uint8 *p);

static void TaskStart(void *pdata);                    // 任务说明
void Task1(void *pdata);
void Task2(void *pdata);

OS_EVENT        *Mbox;                                 // 消息邮箱说明
#define TIMEOUT 20                                     // 定义超时值

/**************************************************************************************
*  函数名称: main()
*  功能描述: 主函数，初始化 μC/OS-II 操作系统，创建任务和消息邮箱，启动系统
*************************************************************************************/
int   main(void){
      BSP_Init();                                  // 硬件目标板初始化
      PINSEL1 = PINSEL1|0x01400000;                // 设置 P0.27、P0.28 连接到 AIN0、AIN1
      AD0CR = (1 << 0)                   |         //  SEL = 1，选择通道 0
              ((Fpclk / 1000000 - 1) << 8) |       //  CLKDIV = Fpclk / 1000000 - 1，
//即转换时钟为 1MHz
              (0 << 16)                  |         //  BURST = 0，软件控制转换操作
              (0 << 17)                  |         //  CLKS = 0，使用 11clock 转换
              (1 << 21)                  |         //  PDN = 1，正常工作模式（非掉电转换模式）
```

```
            (0 << 22)               |       // TEST1:0 = 00，正常工作模式（非测试模式）
            (1 << 24)               |       // START = 1，直接启动 ADC 转换
            (0 << 27);                      // EDGE = 0（CAP/MAT 引脚下降沿触发 ADC 转换）
    OSInit ();
    OSTaskCreate(Task1, (void *)0, &Task1Stk[TASK_STK_SIZE - 1], 3);      //创建任务
    OSTaskCreate(Task2, (void *)0, &Task2Stk[TASK_STK_SIZE - 1], 4);
    OSTaskCreateExt(TaskStart,                                     //任务名
                    (void *)0,                                     //参数指针
                    &TaskStartStk[TaskStkLengh - 1],               //堆栈栈顶
                    TASK_START_PRIO,                               //任务优先级
                    TASK_START_ID,                                 //任务 ID
                    &TaskStartStk[0],                              //堆栈栈底
                    TaskStkLengh,                                  //堆栈长度
                    &TaskUserData[TASK_START_ID],                  //扩展 TCB 数据指针
                    0);                                            //禁止堆栈检测
    Mbox = OSMboxCreate((void *)1);                                //创建消息邮箱
    OSStart ();                                                    //启动 μC/OS-II 多任务调度
    return 0;
}

/*****************************************************************************************
* 函数名称: TaskStart()
* 功能描述: 起始任务，硬件目标板初始化
*****************************************************************************************/
void   TaskStart(void *pdata){
    pdata = pdata;                                  // 避免编译警告
    UART0Init(57600);                               // 串行口初始化
    lcd_init();
    for(;;){
        OSCtxSwCtr = 0;                             //清除上下文切换计数器
        OSTimeDly(70);                              //任务切换，交出 CPU 使用权
    }
}

/*****************************************************************************************
* 函数名称: Task2()
* 功能描述: 启动 A-D 转换并在 LCD 上显示，通过消息邮箱将 A-D 转换数据发送给 Task1
*****************************************************************************************/
void Task2(void *pdata){
    char      str[20];
    uint32    ADC_Data;
    for (;;){
        DisText(0x80,"Task2 sent A-D Dat.");
        DisText(0xc0,"                ");
        AD0CR = (AD0CR&0x00FFFF00)|0x01|(1 << 24);
                                            // 设置 A-D 通道 1，并进行第一次转换
```

```
            while( (AD0DR&0x80000000)==0 );                 // 等待转换结束
            AD0CR = AD0CR | (1 << 24);                      // 再次启运转换
            while( (AD0DR&0x80000000)==0 );                 // 等待转换结束
            ADC_Data = AD0DR;                               // 读取 ADC 结果
            ADC_Data = (ADC_Data>>6) & 0x3FF;               // 提取 A-D 转换值
            ADC_Data = ADC_Data * 3300/1024;                // 数值转换
            sprintf(str, "VIN1=%dmV \r", ADC_Data);
                OSMboxPost(Mbox,str);                       // 向邮箱发送数据
            UartSendString("Task2 sent A-D Dat to Mbox.\r\n\r\n");
            OSTimeDly(400);
        }
}

/*****************************************************************************************
*  函数名称: Task1()
*  功能描述: 从消息邮箱获取 A-D 转换数据
*****************************************************************************************/
void Task1(void *pdata){
    INT8U err;
    char *msg;
    while(1){
        DisText(0x80,"Task1 get A-D Dat.    ");
        DisText(0xc0,"                      ");
         msg=OSMboxPend(Mbox,TIMEOUT,&err);          // 从邮箱获取数据
          if(msg[0]=='V'){
                DisText(0xc0,msg);
                UartSendString("Task1 get A-D Dat from Mbox.\r\n");
                UartSendString(msg);
                UartSendString("\r\n");
          }
         OSTimeDly(160);
    }
}
```

复习思考题

1．μC/OS-II 具有哪些特点？

2．μC/OS-II 的核心代码文件有哪些？

3．μC/OS-II 与处理器相关的文件有哪些？

4．移植 μC/OS-II 通常需要编写哪些文件？

5．μC/OS-II 的任务具有什么特别之处？它能够被应用程序调用吗？

6．μC/OS-II 提供了哪些任务管理函数？

7．编写一个能在 LPC2138 上运行的 μC/OS-II 应用程序，创建两个任务 Task1 和 Task2，并通过 LCD 显示其运行状态。

8．简述 μC/OS-II 信号量有哪两种类型。

9．μC/OS-II 提供了哪些信号量管理函数？

10．编写一个能在 LPC2138 上运行的 μC/OS-II 应用程序，创建两个任务 Task1 和 Task2，它们都要使用 UART0 向外输出信息，创建互斥信号量 semaphore 来进行约束，当 Task1 使用 UART0 时 Task2 就不能使用，反之亦然。

11．简述 μC/OS-II 消息邮箱具有哪些特点。

12．μC/OS-II 提供了哪些消息邮箱管理函数？

13．编写一个能在 LPC2138 上运行的 μC/OS-II 应用程序，创建两个任务 Task1 和 Task2，以及 1 个消息邮箱。在 Task2 中启动 A-D 转换，并通过消息邮箱将 A-D 转换数据发送给 Task1，Task1 从消息邮箱中获取 A-D 转换数据，通过 UART0 向外输出。

附　　录

附录 A　ARM 硬件开发平台

广州风标电子技术有限公司生产的 FB-EDU-LPC2138-A 实验箱，是一款非常适合于 ARM 嵌入式系统基础教学的硬件开发平台，本书前 6 章提及的 Proteus 仿真例子，都可以在该硬件平台上予以实现。开发平台主要包含如下硬件电路模块：I^2C EEPROM 电路模块、SD 卡电路模块、音频功放电路模块、ADC 电路模块、8 个发光二极管、8 个独立按键电路模块、4 个静态数码管、LCD128×128 液晶模块、ENC28J60 10M 以太网络电路模块、SL811 USB 接口电路模块、MODEM 接口电路模块、RS-232 总线接口、RS-485 总线接口、CAN 总线接口、PS/2 键盘接口、直流电动机和电动机测速模块、步进电动机驱动模块、功能扩展模块等，各模块在实验箱上的位置如图 A-1 所示。

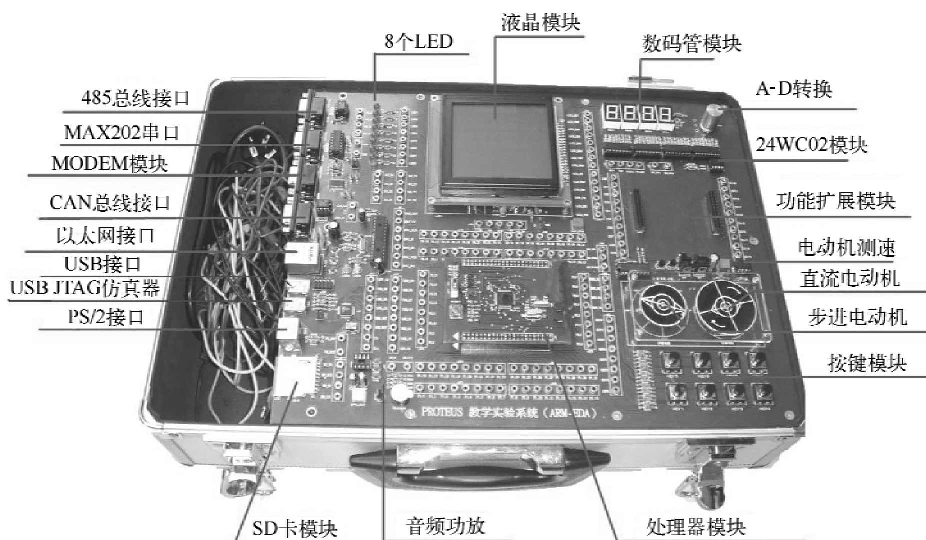


图 A-1　硬件电路模块

A.1　电源电路

本教学实验开发平台采用开关电源供电，开关电源具有 12V、5V、−5V 电源输出接口，开关电源具有短路保护，过载保护等功能，具有较强的带负载能力。系统中部分电路模块如电动机模块是 5V 供电，而 LPC2138 是 3.3V 单电源供电，其大多外围部件也是 3.3V 供电。5V 电源由开关电源直接提供，5V 通过 LDO 芯片稳压输出 3.3V 电压。

稳压芯片采用 SPX117M3-3.3V，其特点是输出电流可达 800mA，输出电压精度在±1%以内，还具有电流限制和热保护功能。使用时输出端应至少接一个 10μF 以上的电容来改善瞬态响应和稳定性。POWER 是电源指示灯。电源电路如图 A−2 所示。

图 A−2　电源电路

A.2　复位电路

LPC2138 芯片具有高速、低功耗和低工作电压等特性，导致其噪声容限低，对电源的纹波、瞬态性能、电源监控可靠性等诸多方面提出了更高的要求。

本教学实验开发平台采用了专用微处理器电源监控芯片 SP708 设计复位电路，提高了系统的可靠性。复位电路如图 A−3 所示。

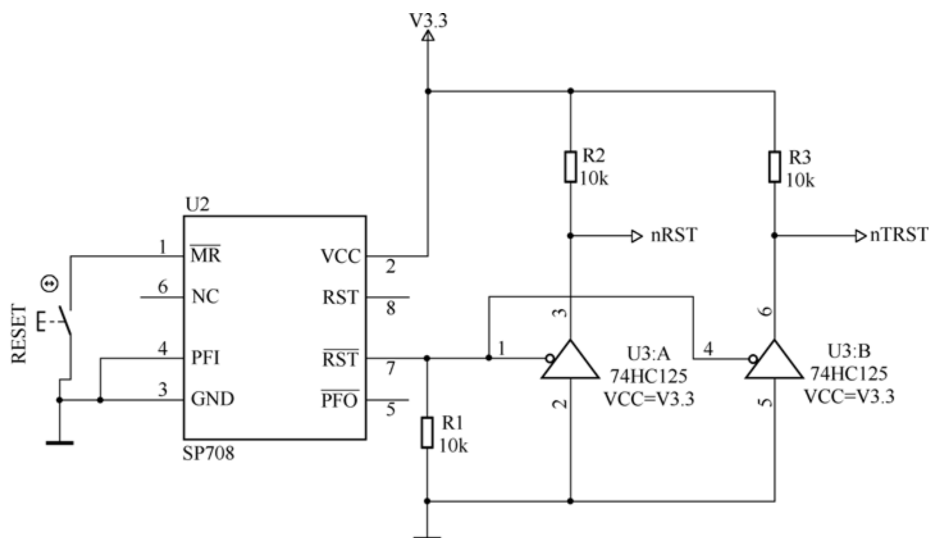


图 A−3　复位电路

A.3　时钟电路

LPC2138 系统时钟可使用外部晶振或外部时钟源，时钟频率为 10～25MHz，内部 PLL 电路可对系统时钟进行倍频，使系统运行速度更快（CPU 最大操作时钟为 60MHz）。RTC 时钟由独立的 32.768kHz 晶体振荡器或基于 VPB 时钟的可编程预分频器来提供。本教学实验

开发平台的系统时钟采用 25M 外部晶振，RTC 时钟采用独立的 32.768kHz 晶振。时钟电路如图 A-4 所示。

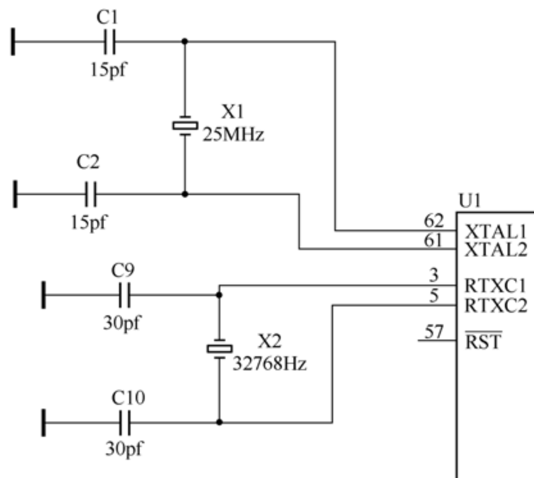


图 A-4　时钟电路

A.4　RS232 串口和 MODEM 电路

本教学实验开发平台的接口电压为 3.3V，为实现与 PC 进行通信，采用了 MAX202 进行 RS232 电平转换，MAX202 是 3V 工作电压的 RS232 转换芯片，发光管 RXD 和 TXD 用于指示串口的通信状态。另外，LPC2138 的 UART1 带有完全的调制解调器接口，具有 3 路发送和 5 路接收信道，所以要使用 8 路 RS232 转换芯片 SP3243，构成一个完整的 MODEM 接口电路。RS232 串口和 MODEM 电路如图 A-5 所示，在进行 ISP 下载时，需将 J46 用跳线帽进行短接。

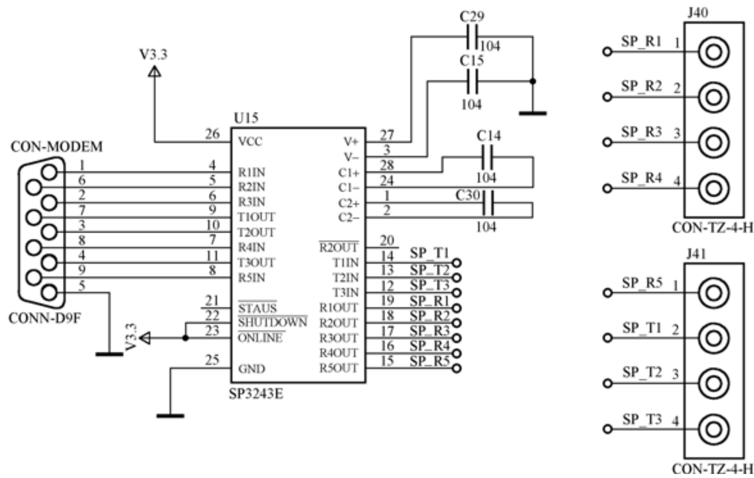


图 A-5　RS232 串口和 MODEM 电路

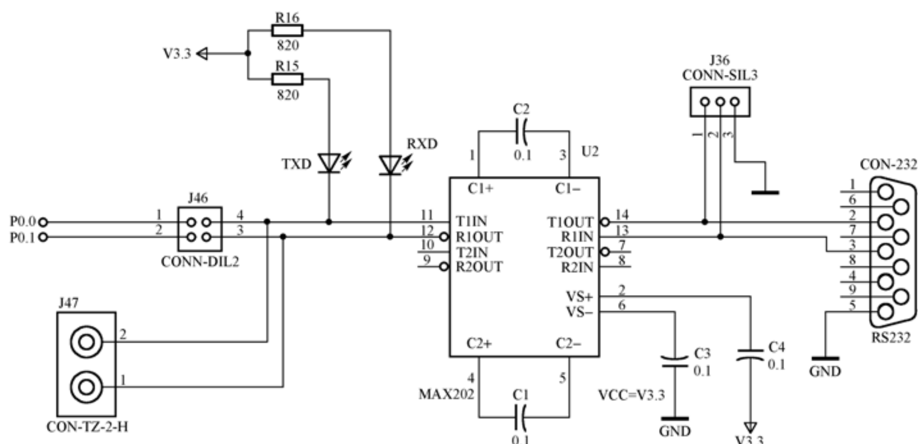


图 A-5　RS232 串口和 MODEM 电路（续）

A.5　键盘电路

本教学实验开发平台配有 8 个独立按键，分别为 KEY1～KEY8，如图 A-6 所示。由于 GPIO 作为输入时，内部无上拉电阻，所以要使用 R71～R78 8 个上拉电阻，没有按键揿下时，口线值为 1，当有按键揿下时，口线值为 0。R79～R82、R85～R89 为口线保护电阻。

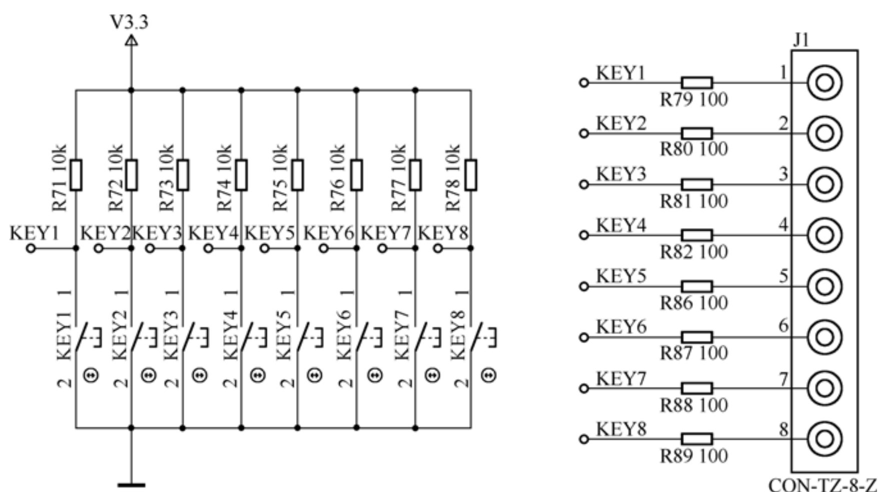


图 A-6　键盘电路

A.6　LED 显示电路

本教学实验开发平台的数码管显示电路采用了 4 片 74HC595 来驱动 4 个静态共阴 LED 数码管，74HC595 具有 8 位移位寄存器和一个存储器，有三态输出功能。输出寄存器可以直接清除，具有 100MHz 的移位频率。数码管显示电路如图 A-7 所示。

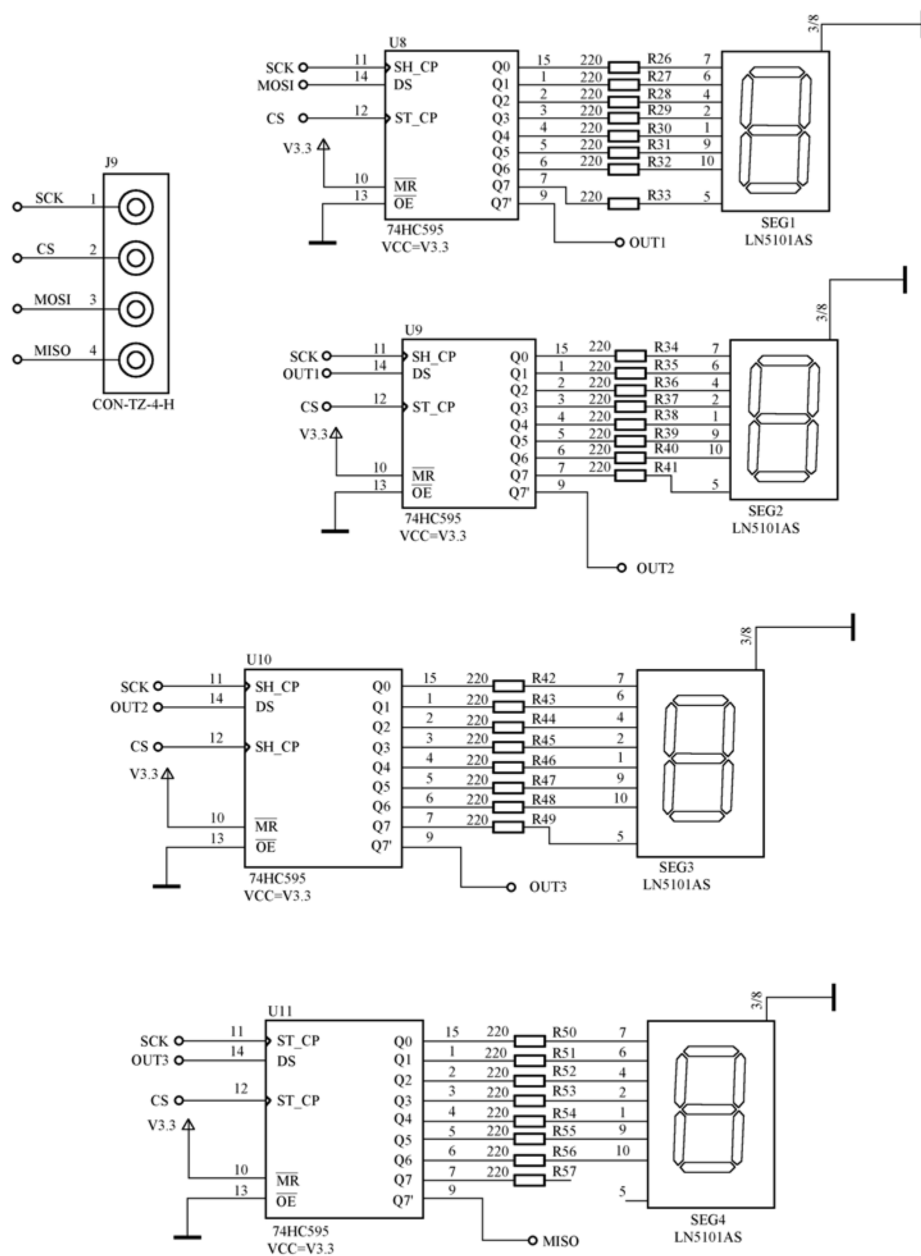


图 A-7　数码管显示电路

本教学实验开发平台配有 8 个独立的 LED 发光二极管 VD4～VD11，R18～R25 为限流电阻，控制 LED 的亮度。R18～R25 一端连接到 J2，J2 输出 1 时，对应的 LED 熄灭，J2 输出 0 时，对应的 LED 点亮，LED 显示电路如图 A-8 所示。

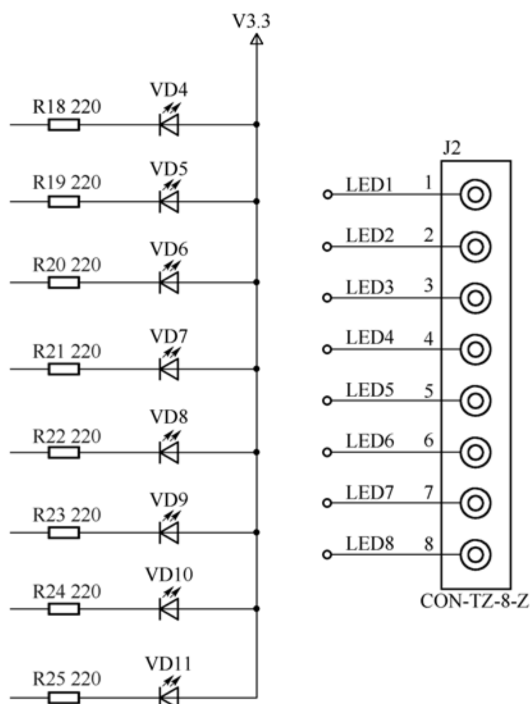


图 A−8　LED 显示电路

A.7　LCD 显示电路

本教学实验开发平台采用 OCM128×128 黑白液晶模块，该液晶模块为 3.3V 供电，不带中文字库，驱动控制器采用 T6963，具有可调对比度。LCD 显示接口电路如图 A−9 所示，图中 LCD_ADJ 是个 50K 的电位器，用来调节液晶屏的对比度。

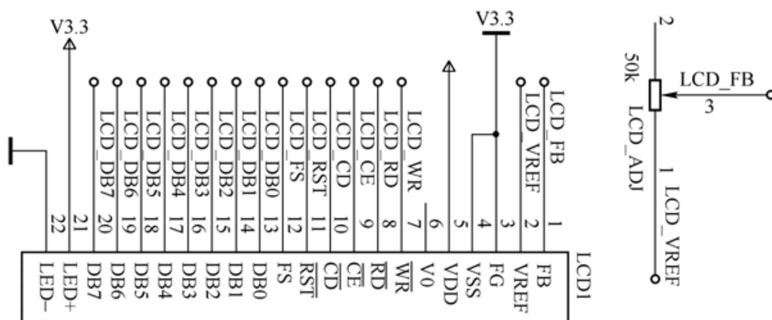


图 A−9　LCD 显示接口电路

A.8　I²C 接口电路

LPC2138 具有支持 400K 高速模式的硬件 I²C 接口，本教学实验开发平台的 I²C 接口采用一片 2K 位的 EEPROM 只读存储器 24WC02，实现 I²C 读写操作实验。具体电路如图 A−10 所示。

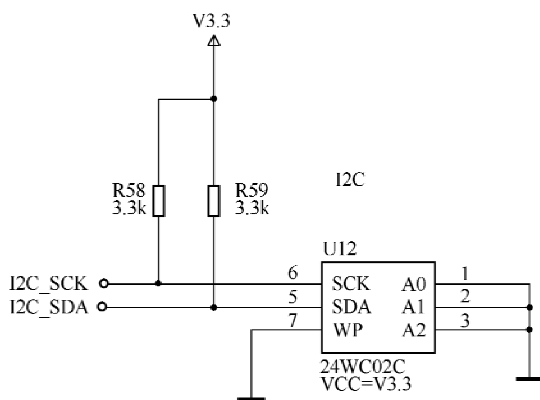


图 A-10　I²C 接口电路

A.9　音频接口电路

本教学实验开发平台配有一个 5V 直流蜂鸣器，采用 NPN 晶体管 2N551 进行驱动，另外还配有一个音频功放电路，采用专用音频 IC 芯片 LM386 进行功率放大。在实验箱配件中配有一个 8Ω、0.5W 的喇叭 BUZZER，可用作音频功放实验。具体电路如图 A-11 所示。

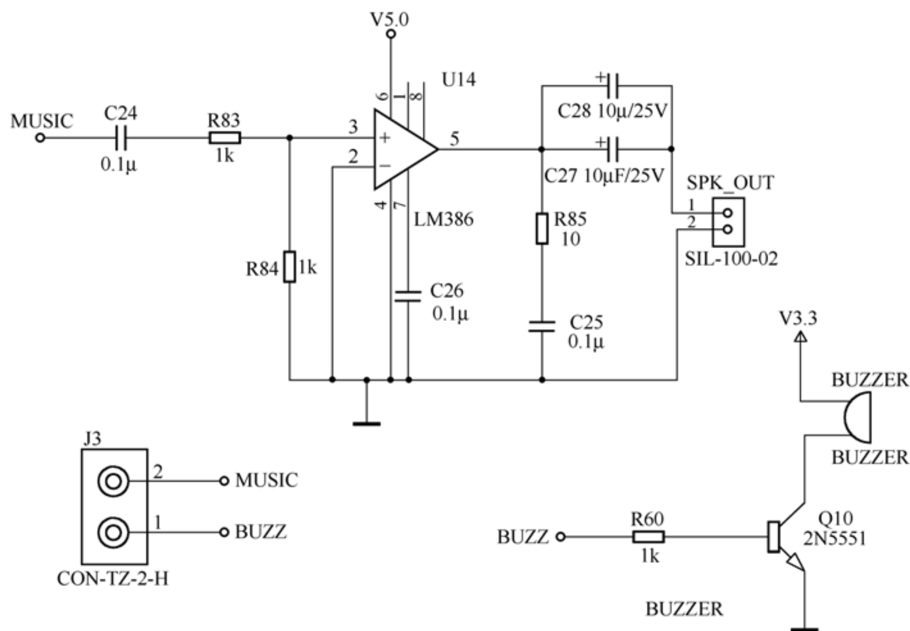


图 A-11　音频接口电路

A.10　电动机驱动电路

本教学实验开发平台配有一个步进电动机，步进电动机要求具有较强的驱动能力，需要专门的驱动电路，本系统采用达林顿晶体管 ULN2003 进行驱动，ULN2003 具有电流增益

高、温度范围宽、带负载能力强等特点，适应于各类要求高速大功率驱动的系统。步进电动机驱动电路如图 A-12 所示。

图 A-12　步进电动机驱动电路

本教学实验开发平台还配有一个直流电动机，采用功率晶体管进行驱动，功率管采用 NPN D669 和 PNP 649 配对使用，两功率管具有 160V 的耐压值，1.5A 的额定电流，20W 的功率驱动能力。直流电动机驱动电路如图 A-13 所示。

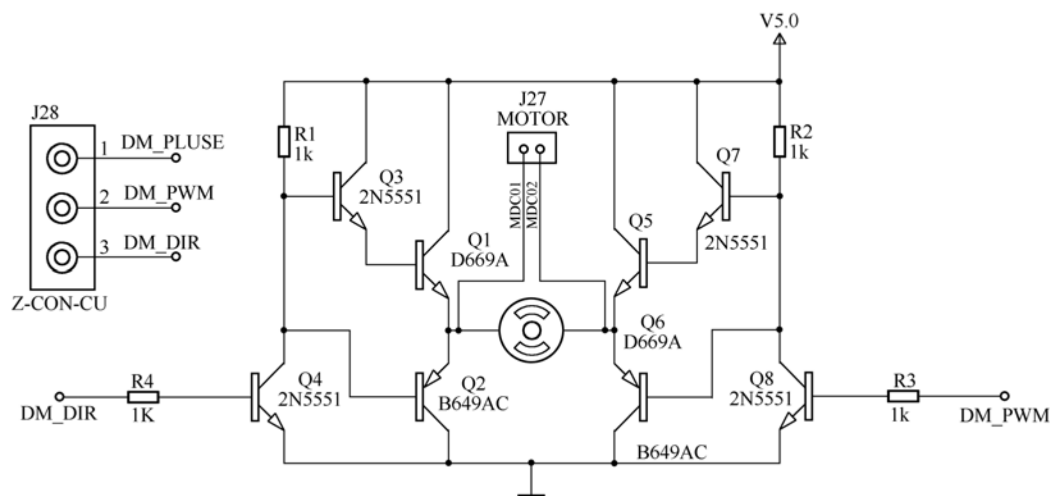


图 A-13　直流电动机驱动电路

直流电动机配有一个测速电路，采用光电对管进行测速，直流电动机叶片底下贴有白色胶纸，每次叶片转过胶纸上时将产生一个脉冲，通过对脉冲数的计数便可达到测速的目的。由于光电对管输出信号较弱，故后级采用一个比较器对弱信号进行放大，比较器采用 LM358 芯片，另外还有一个可调电位器用来调节光电对管的灵敏度。直流电动机测速电路如图 A-14 所示。

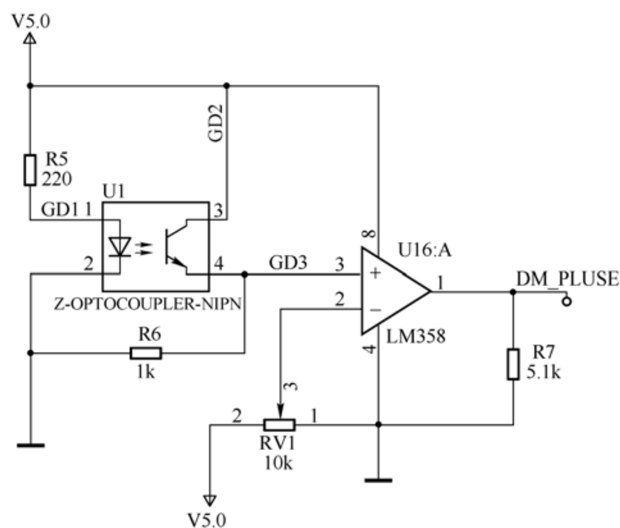


图 A−14　直流电动机测速电路

A.11　ADC 采样电路

本教学实验开发平台配有一路 ADC 采样电路，由电位器串接一个电阻而成，串接电阻是为了避免 ADC 引脚和地直接短接，采样电压范围 0~3.3V。ADC 采样电路如图 A−15 所示。

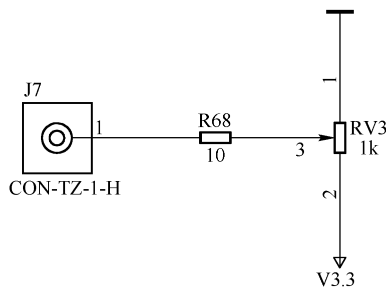


图 A−15　ADC 采样电路

A.12　网络接口电路

本教学实验开发平台配有一个网络接口，可通过网络对实验箱进行访问和数据交换。网络接口采用专用以太网控制芯片 ENC28J60，ENC28J60 是美国 Microchip 公司近期推出的 28 引脚带 SPI 接口的独立以太网控制器，它采用了一系列过滤机制对传输数据包进行限制，还提供了一个内部 DMA 模块，以实现快速数据吞吐和硬件支持的 IP 校验和计算。ENC28J60 与主 CPU 的通信通过两个中断引脚和 SPI 接口实现，数据传输速率高达 10Mbit/s。另外还提供两个专用 LED 引脚用于指示网络活动状态。

设计 NC28J60 硬件接口时，需要注意复位电路、时钟振荡器、振荡器启动定时器、时钟输出引脚、变压器、输入/输出电平等几个方面。网络接口电路如图 A−16 所示。

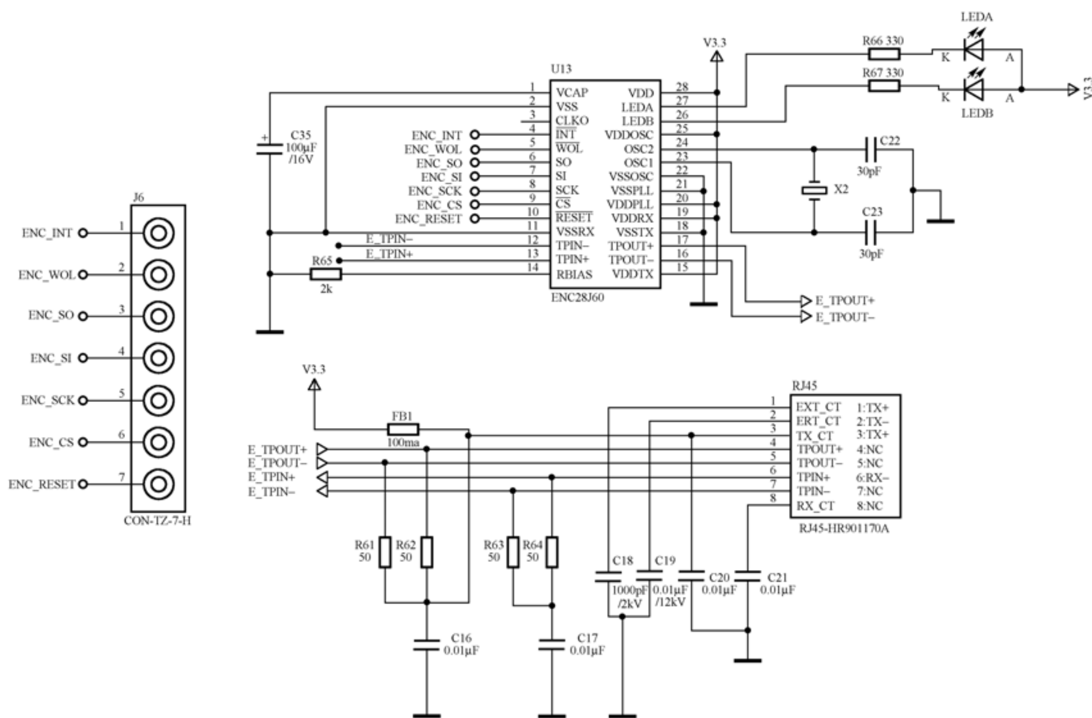


图 A−16　网络接口电路

A.13　SD 卡接口电路

本教学实验开发平台具有读写 SD 卡的功能，电路原理图如图 A−17 所示。SD 卡可以用 SD 总线访问，也可以选用 SPI 总线访问。考虑到 LPC2138 微控制器有 SPI/SSP 接口而没有 SD 总线接口，本系统采用了 SPI 接口。

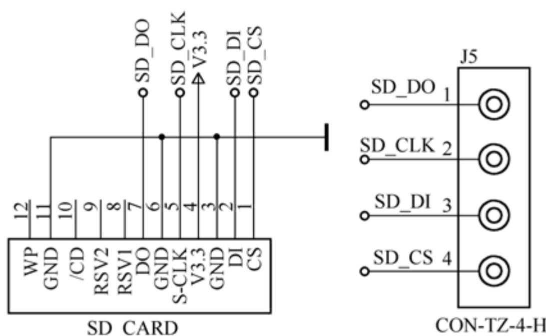


图 A−17　SD 卡接口电路

A.14　USB 接口电路

本教学实验开发平台配有 USB 接口，因为 LPC2138 芯片不具备 USB 接口功能，所以

需外部扩展 USB 控制器，才能实现 USB 传输，本系统采用 USB 控制器芯片 SL811HS。

SL811HS 主要由 CPU 接口、中断控制、主/从模式控制、USB SIE、时钟模块、SRAM 及根集线器等模块组成。SL811HS 符合 USB1.1 标准，可通过软件或硬件选择以主模式和从模式工作，本系统采用硬件对两种模式进行选择。通过对 J35 跳线帽的位置设置来选择 USB 为主模式还是从模式，当 USB_MS、PD1、PD2 与 GND 短接时为主模式，当 USB_MS、PU 与 3.3V 短接时为从模式，在从模式下还支持 DMA 传输方式。

SL811HS 需要一个时钟电路，本系统采用 48M 的有源晶振，为了向 SL811 芯片提供足够幅值的振荡信号，晶振采用 5V 供电。USB 接口电路如图 A−18 所示。

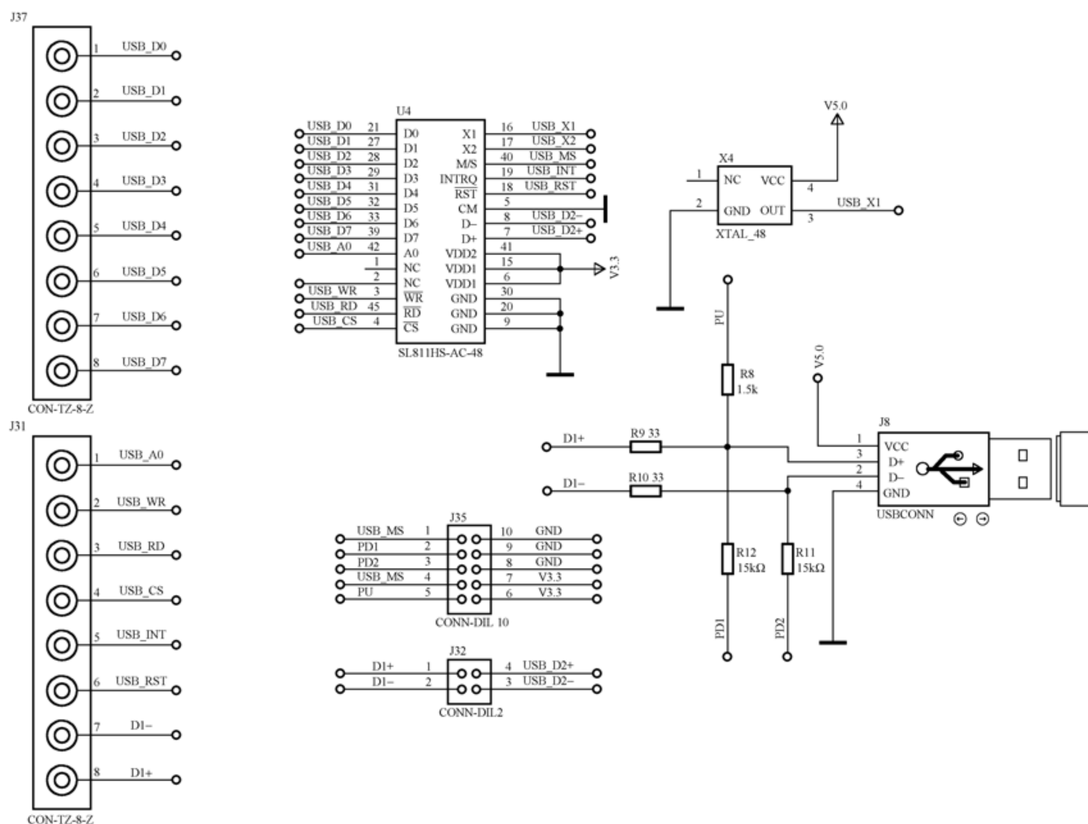


图 A−18　USB 接口电路

A.15　总线接口电路

本教学实验开发平台配有 485 总线接口和 CAN 总线接口。485 总线接口采用 MAX485 芯片，CAN 总线采用 82C250 芯片，这两种芯片都是 5V 供电（LPC2138 的 I/O 可承受 5V 电压），总线接口电路如图 A−19 所示。

A.16　PS/2 接口电路

本教学实验开发平台配有 PS/2 鼠标、键盘接口，其电路如图 A−20 所示。

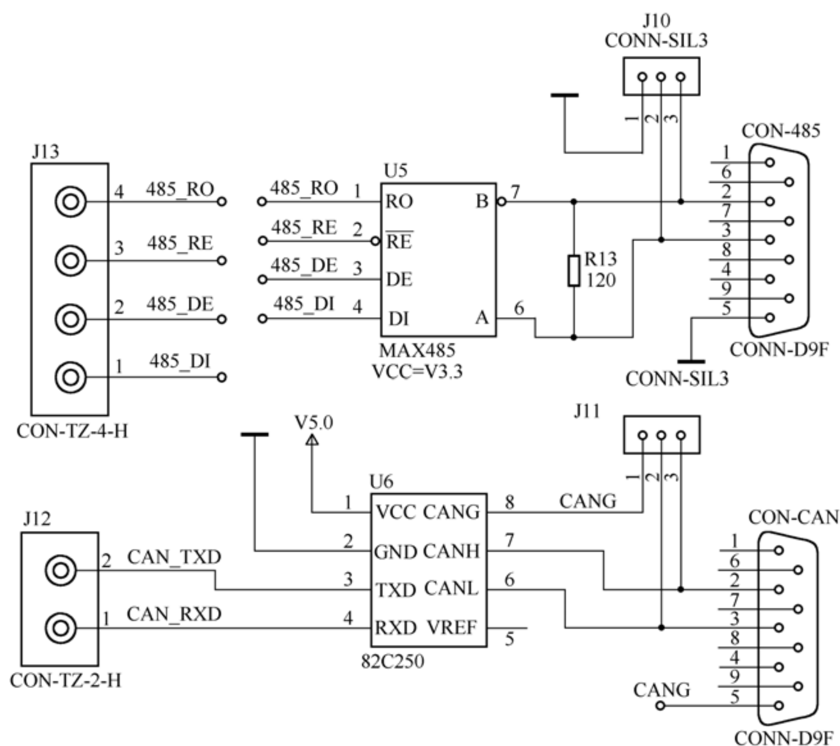


图 A-19　总线接口电路

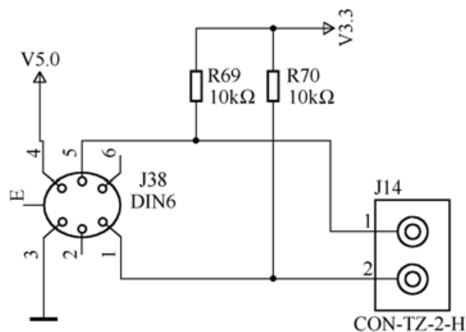


图 A-20　PS/2 接口电路

A.17　扩展模块接口电路

本教学实验开发平台配有扩展接口模块，方便学生在实验箱上做一些扩展实验。扩展接口模块有 20 个 I/O 口，具有 3.3V 和 5V 两种电源。扩展接口模块电路如图 A-21 所示。

A.18　JTAG 接口电路

本教学实验开发平台采用 ARM 公司提出的标准 20 脚 JTAG 调试接口，如图 A-22 所示。图中 JTAG 接口上的信号 nTRST 连接到 LPC2138 芯片的 TRST 引脚，以达到控制

LPC2138 内部接口电路复位的目的（但不控制 CPU 复位）。在 RTC 引脚接一个 4.7kΩ的下拉电阻，目的是系统复位后 LPC2138 内部 JTAG 接口使能，这样就可以直接进行硬件仿真调试了。如果用户需要将 P1.26～P1.31 作为 I/O 端口使用，不进行 JTAG 仿真调试，则可以在用户程序中通过设置 PINSEL2 寄存器，对 LPC2210 内部 JTAG 接口进行禁止。

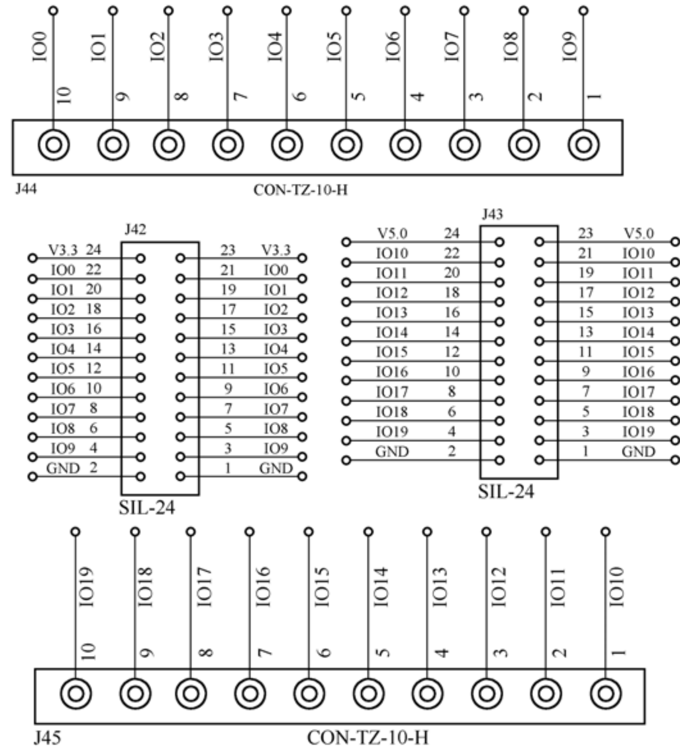


图 A-21　扩展接口模块电路

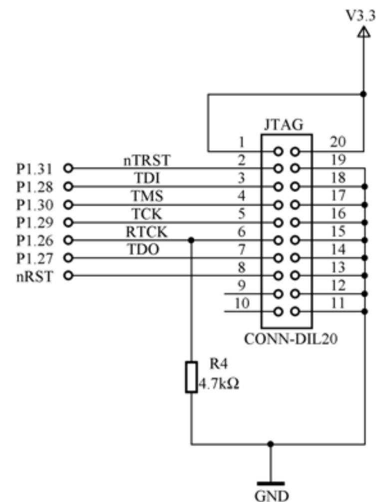


图 A-22　JTAG 接口电路

附录 B　Proteus 中的常用元器件

元 器 件 名	中 文 注 释	元 器 件 名	中 文 注 释
LPC2138	ARN7 处理器	RELAY	继电器
LM3S328	ARM Cortex M3 处理器	ALTERNATOR	交互式交流电压源
CRYSTAL	晶体振荡器	POT-LIN	交互式电位计
CERAMIC22P	陶瓷电容	CAP-VAR	可变电容
CAP	电容	CELL	单电池
CAP-ELEC	通用电解电容	BATTERY	电池组
RES	电阻	AREIAL	天线
RX8	8 电阻排	PIN	单脚终端接插针
RESPACK-8	带公共端的 8 电阻排	LAMP	动态灯泡模型
MINRES5K6	5K6 电阻	TRAFFIC	动态交通灯模型
74LS00	四 2 输入与非门	SOUNDER	压电发声模型
74LS164	8 位并出串行移位寄存器	SPEAKER	喇叭模型
74LS244	8 同相三态输出缓冲器	7805	5V，1A 稳压器
74LS245	8 同相三态输出收发器	78L05	5V，100mA 稳压器
NOR	二输入或非门	LED-GREEN	绿色发光二极管
OR	二输入或门	LED-RED	红色发光二极管
XOR	二输入异或门	LED-YELLOW	黄色发光二极管
NAND	二输入与非门	MAX7219	串行 8 位 LED 显示驱动器
AND	二输入与门	7SEG-BCD	七段 BCD 数码管
NOT	数字反相器	7SEG-DIGITAL	七段数码管
COMS	COMS 系列	7SEG-COM-CAT-GRN	七段共阴极绿色数码管
4001	双 2 输入或非门	7SEG-COM-AN-GRN	七段共阳极绿色数码管
4052	双 4 通道模拟开关	7SEG-MPX6-CA	6 位七段共阳极红色数码管
4511	BCD-7 段锁存/解码/驱动器	7SEG-MPX6-CC	6 位七段共阴极红色数码管
DIODE-TUN	通用沟道二极管	MATRIX-5×7-RED	5×7 点阵红色 LED 显示器
UF4001	二极管急速整流器	MATRIX-8×8-BLUE	8×7 点阵蓝色 LED 显示器
1N4148	小信号开关二极管	AMPIRE128×64	128×64 图形 LCD
SCR	通用晶闸管整流器	LM016L	16×2 字符 LCD
TRIAC	通用三端双向晶闸管开关	555	定时器/振荡器
MOTOR	简单直流电动机	NPN	通用 NPN 型双极性晶体管
MOTOR-STEPPER	动态单极性步进电动机	PNP	通用 PNP 型双极性晶体管

（续）

元 器 件 名	中 文 注 释	元 器 件 名	中 文 注 释
MOTOR-SERVO	伺服电动机	PMOSFET	通用 P 型金属氧化物场效应管
COMPIN	COM 口物理接口模型	2764	8K×8 EPROM 存储器
CONN-D9M	9 针 D 型连接器	6264	8K×8 静态 RAM 存储器
CONN-D9F	9 孔 D 型连接器	24C04	4K 位 I2C EEPROM 存储器
BUTTON	按钮	ADC0808	8 位 8 通道 A-D 转换器
SWITCH	带锁存开关	DAC0832	8 位 D-A 转换器
SW-SPST-MOM	非锁存开关	DS1302	日历时钟

参 考 文 献

[1] ARM. ARM7TDMI-S Technical Reference Manual. www.arm.com. 2001

[2] D Seal. ARM Architecture Reference Manual[M]. 2nd ed. New Jersey: Addison-Wesley, 2000.

[3] LPC213x User Manual. NXP Semiconductors. 2010

[4] LPC2138 Errata Sheet V2 . NXP Semiconductors. 2011

[5] μC/OS-II Time Management in LPC2000. NXP Semiconductors. 2005.

[6] A N Sloss, D Symes, C Wright. ARM System Developer’s Guide[M]. California: Morgan Kaufman, 2004.

[7] 周立功. 深入浅出 ARM7－LPC213x/214x[M]. 北京：北京航空航天大学出版社，2005.

[8] Jean J Labrosse. 嵌入式实时操作系统 μC/OS-II[M]. 邵贝贝, 等译. 2 版. 北京：北京航空航天大学出版社，2003.

[9] 张勇. 嵌入式操作系统原理与面向任务程序设计[M]. 西安：西安电子科技大学出版社，2010.

[10] 徐爱钧. IAR EWARM V5 嵌入式系统应用编程与开发[M]. 北京：北京航空航天大学出版社，2009.

[11] 徐爱钧. 单片机原理与应用——基于 Proteus 虚拟仿真技术[M]. 北京：机械工业出版社，2010.

[12] 周润景, 袁伟亭. 基于 Proteus 的 ARM 虚拟开发技术[M]. 北京：北京航空航天大学出版社，2007.

[13] 范书瑞, 赵燕飞, 高铁成. ARM 处理器与 C 语言开发应用[M]. 北京：北京航空航天大学出版社，2008.

[14] 张瑜, 王益涵. ARM 嵌入式程序设计[M]. 北京：北京航空航天大学出版社，2009.

[15] 任哲. ARM 体系结构及其嵌入式处理器[M]. 北京：北京航空航天大学出版社，2008.

[16] 沈连丰, 等. 嵌入式系统及其开发应用[M]. 北京：电子工业出版社，2005.

[17] 桂电-丰宝联合实验室. ARM 原理与嵌入式应用——基于 LPC2400 系列处理器和 IAR 开发环境[M]. 北京：电子工业出版社，2008.

[18] 吴明辉. 基于 ARM 的嵌入式系统开发与应用[M]. 北京：人民邮电出版社，2004.



本科电气精品教材推荐

电气信息工程丛书

西门子工业通信网络组态编程与故障诊断

书号：28256　　定价：69.00 元

作者：廖常初　　配套资源：DVD 光盘

推荐简言：

本书建立在大量实验的基础上，详细介绍了实现通信最关键的组态和编程方法，随书光盘有上百个通信例程，绝大多数例程经过硬件实验的验证。读者根据正文介绍的通信系统的组态步骤和方法，参考光盘中的例程作组态和编程练习，可以较快地掌握网络通信的实现方法。

西门子 PLC 高级应用实例精解

书号：29304　　定价：42.00 元

作者：向晓汉　　配套资源：DVD 光盘

推荐简言：

本书通过实例全面讲解西门子 S7-200/S7-1200/S7-300 PLC 的高级应用。内容包括梯形图的编程方法、PLC 在过程控制中应用、PLC 在运动控制中的应用、PLC 的通信及其通信模块的应用等。书中实例都用工程实际的开发过程详细介绍，便于读者模仿学习。每个实例都有详细的软件、硬件配置清单，并配有接线图和程序。本书所附配套资源中有重点实例源程序和操作过程视频文件。

西门子 WinCC V7 基础与应用

书号：32902　　定价：44.00 元

作者：甄立东　　配套资源：DVD 光盘

推荐简言：本书系统地介绍了 WinCC V7.0 的功能及其组态方法。首先介绍了初级用户必须掌握的主要功能，其次介绍了高级用户需要了解的 Microsoft SQL Server 2005、冗余系统组态、全集成自动化、开发性和工厂智能选件。通过实例，详尽地展示了各种应用的设计和实现步骤以及应用。本书还对 WinCC V7.0 新增功能进行了详细讲解。

现场总线与工业以太网及其应用技术

书号：35607　　定价：58.00 元

作者：李正军　　配套资源：电子教案

推荐简言：本书从工程实际应用出发，全面系统地介绍了现场总线与工业以太网技术及其应用系统设计，力求所讲内容具有较强的可移植性、先进性、系统性、应用性、资料开放性，起到举一反三的作用。主要内容包括现场总线与工业以太网概论、控制网络技术、通用串行通信接口技术、PROFIBUS 现场总线、PROFIBUS-DP 通信控制器与网络接口卡等。

PLC 编程及应用 第 3 版

书号：10877　　定价：　37.00 元

作者：廖常初　　配套资源：DVD 光盘

获奖情况：全国优秀畅销书

推荐简言：西门子公司重点推荐图书，累计销量已达 12 万册。本书以西门子公司的 **S7-200 PLC** 为例，介绍了 **PLC** 的工作原理、硬件结构、指令系统、最新版编程软件和仿真软件的使用方法；介绍了数字量控制梯形图的一整套先进完整的设计方法；介绍了 **S7-200** 的通信网络、通信功能和通信程序的设计方法等。配套光盘有 **S7-200** 编程软件和 **OPC** 服务器软件 **PC Access**、与 **S7-200** 有关的中英文用户手册和资料、应用例程等

S7-300/400 PLC 应用技术 第 3 版

书号：36379　　定价：69.00 元

作者：廖常初　　配套资源：DVD 光盘

推荐简言：西门子公司重点推荐图书，销量已达 8 万册。本书介绍了 S7-300/400 的硬件结构、性能指标和硬件组态的方法；指令系统、程序结构、编程软件 STEP7 的使用方法；梯形图的经验设计法、继电器电路转换法和顺序控制设计法，以及使用顺序功能图语言 S7 Graph 的设计方法。另外还介绍了 S7-300/400 的网络结构，AS-i 和工业以太网、PRODAVE 通信软件的组态、参数设置的编程的方法。配套的光盘附有大量的中英文用户手册、软件和例程，附有 STEP7 编程软件。

机工出版社·计算机分社书友会邀请卡

尊敬的读者朋友：

感谢您选择我们出版的图书！我们愿以书为媒与您做朋友！我们诚挚地邀请您加入：

“机工出版社·计算机分社书友会”

以书结缘，以书会友

加入“书友会”，您将：

★ 第一时间获知新书信息、了解作者动态；

★ 与书友们在线品书评书，谈天说地；

★ 受邀参与我社组织的各种沙龙活动，会员联谊；

★ 受邀参与我社作者和合作伙伴组织的各种技术培训和讲座；

★ 获得“书友达人”资格（积极参与互动交流活动的书友），参与每月 5 个名额的“书友试读赠阅”活动，获得最新出版精品图书 1 本。

如何加入“机工出版社·计算机分社书友会”

两步操作轻松加入书友会

Step1

访问以下任一网址：

★ 新浪官方微博：http://weibo.com/cmpjsj

★ 新浪官方博客：http://blog.sina.com.cn/cmpbookjsj

★ 腾讯官方微博：http://t.qq.com/jigongchubanshe

★ 腾讯官方博客：http://2399929378.qzone.qq.com

Step2

找到并点击调查问卷链接地址（通常位于置顶位置或公告栏），完整填写调查问卷即可。

联系方式

通信地址：北京市西城区百万庄大街 22 号
机械工业出版社计算机分社

邮政编码：100037

联系电话：010-88379750

传　　真：010-88379736

电子邮件：cmp_itbook@163.com

敬请关注我社官方微博：　http://weibo.com/cmpjsj

第一时间了解新书动态，获知书友会活动信息，与读者、作者、编辑们互动交流！

在线互动交流平台

官方微博：http://weibo.com/cmpjsj

豆 瓣 网：http://site.douban.com/139085/

读者信箱：cmp_itbook@163.com

◇ PLC 编程及应用（第3版）

◇ S7-300/400 PLC应用技术（第3版）

◇ 西门子S7-200 PLC编程及应用案例精选

◇ 西门子人机界面(触摸屏)组态与应用技术

◇ 西门子工业通信网络组态编程与故障诊断

◇ 西门子 WinCC V7基础与应用

◇ 三菱PLC工程应用设计

◇ CS/CJ系列PLC应用基础及案例

◇ 欧姆龙CP1H PLC应用基础与编程实践

◇ IEC 61131-3 编程语言及应用基础

◇ 网际组态软件Advantech WebAccess应用技术

◇ 组态软件基础与工程应用(易控INSPEC)

◇ Protel 99 SE原理图与PCB设计及电路仿真

◇ Protel 99 SE实战详解与技巧

◇ Protel 2004电路原理图及PCB设计

◇ 基于Altium Designer的原理图与PCB设计

◇ 基于Allegro的PCB设计与开发

◇ VHDL数字电路及系统设计

◇ FPGA/VHDL设计入门与进阶

◇ OrCAD电路原理图设计与应用

◇ 单片机应用及51程序设计（第2版）

◇ 51单片机快速上手

◇ 数字信号处理器原理、结构及应用基础——TMS320F28x

◇ DSP技术及应用系统设计

◇ Freescale 9S12十六位单片机原理及嵌入式开发技术

◇ 基于EDK的FPGA嵌入式系统开发

◇ 基于模型的设计——Qsys篇

◆ ARM嵌入式系统基础与应用—基于Proteus和IAR EWARM

◇ 嵌入式可配置实时操作系统eCos开发与应用（第2版）

◇ Linux PowerPC详解——核心篇

◇ 典型数控系统应用技术（FANUC篇）

◇ 现场总线控制网络技术

◇ 现场总线与工业以太网及其应用技术

◇ 数控原理与编程

◇ 风电并网运行与维护

◇ 无线传感器网络及其在物流中的应用

◇ CAN总线技术与应用系统设计

地址：北京市百万庄大街22号
邮政编码：100037
电话服务
社服务中心：010-88361066
销售一部：010-68326294
销售二部：010-88379649
读者购书热线：010-88379203
网络服务
教材网：http://www.cmpedu.com
机工官网：http://www.cmpbook.com
机工官博：http://weibo.com/cmp1952
封面无防伪标均为盗版

上架指导 工业技术/嵌入式

ISBN 978-7-111-44508-1

策划编辑◎时静 / 封面设计◎刘吉维



定价：56.00元（含1DVD）